Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Институт радиоэлектроники и информационных технологий — РТФ

Ю. П. ПАРФЁНОВ

ПОСТРЕЛЯЦИОННЫЕ ХРАНИЛИЩА ДАННЫХ

Учебное пособие

Министерство образования и науки Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

Ю. П. Парфёнов

Постреляционные хранилища данных

Учебное пособие

Рекомендовано
методическим советом УрФУ для студентов,
обучающихся по программе магистратуры
по направлению подготовки «Информатика
и вычислительная техника»

Екатеринбург
Издательство Уральского университета
2016

УДК 004.65(075.8)
ББК 32.973я73
П18

Рецензенты:
кафедра математики и естественно-научных дисциплин Уральского института экономики, управления и права (завкафедрой канд. физ.-мат. наук, доц. С. П. Трофимов);
руководитель проекта в УБРиР канд. экон. наук Н. А. Бегунов
Научный редактор — канд. пед. наук, доц. Н. В. Папуловская

Парфенов, Ю. П.

П18 Постреляционные хранилища данных : учеб. пособие / Ю. П. Парфенов. — Екатеринбург : Изд-во Урал. ун-та, 2016. — 120 с.

ISBN 978-5-7996-1827-8

Учебное пособие предназначено для подготовки магистрантов по направлению «Информатика и вычислительная техника» по профилям «Информационно-управляющие системы» и «Компьютерный анализ и интерпретация данных». В пособии рассматриваются требования к хранилищам данных в условиях информационной глобализации. Приводятся классификация новых типов хранилищ, характеристика используемых моделей данных и методов их обработки. Дается описание приемов работы с объектно-реляционными и NoSQL базами данных. Рассматриваются методы и средства хранения и обработки больших данных.

Библиогр.: 21 назв. Табл. 4. Рис. 31.

УДК 004.65(075.8)
ББК 32.973я73

Учебное издание

Парфёнов Юрий Павлович

ПОСТРЕЛЯЦИОННЫЕ ХРАНИЛИЩА ДАННЫХ

Подписано в печать 04.10.2016. Формат 60×84/16. Бумага писчая. Печать цифровая.
Гарнитура Newton. Уч.-изд. л. 5,4. Усл. печ. л. 7,0. Тираж 50 экз. Заказ 334

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5. Тел.: 8(343)375-48-25, 375-46-85, 374-19-41.
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620075, Екатеринбург, ул. Тургенева, 4. Тел.: 8(343) 350-56-64, 350-90-13. Факс: 8(343) 358-93-06

ISBN 978-5-7996-1827-8

Оглавление

Основные сокращения

HDFS	– Hadoop Distributed File System
HDMR	– Hadoop MapReduce
JSON	– JavaScript Object Notation
NoSQL	– Not Only SQL
ODMG	– Object Data Management Group
БД	– база данных
ИС	– информационная система
ИТ	– информационные технологии
КИС	– корпоративная информационная система
ООБД	– объектно-ориентированная база данных
РБД	– реляционная база данных
РФС	– распределенная файловая система
СУБД	– система управления базой данных
ХД	– хранилище данных

1. Условия появления новых типов хранилищ данных

1.1. Рост объема информации — реалии цифровой вселенной

Необычайное влияние ИТ на все сферы жизни общества начиная с последней четверти XX века породило метафору «информационная революция». Современная (с 2000 г.) пятая, следующая за письменностью, книгопечатанием, телефонией и радиосвязью, ЭВМ и персональными компьютерами, революция объединяет и синергически усиливает эффекты предшествующих изобретений и технических решений в области хранения, передачи и обработки информации. Современные информационные технологии и Интернет обеспечивают автоматическое накопление и обмен информацией как в масштабах отдельного человека, компании, так и всего человечества. Доступ ко всей массе накапливаемых данных и возможность ее автоматизированной обработки меняет стиль жизни в информационном обществе, умножает его интеллектуальные способности. Стремительное увеличение числа источников, создающих цифровые данные, приводит к взрывному росту объема накапливаемой в мире информации и создает новые проблемы ее хранения и обработки.

По оценкам компании Linxdatacenter, с одной стороны, наблюдается стремительный рост объема корпоративных данных и их ценности для принятия решений. С другой стороны, этот рост вызывает усложнение задач построения эффектив-

ной и безопасной ИТ-среды для хранения, передачи и обработки данных. Процент информации, нуждающейся в защите, неуклонно растет, в то же время уровень защиты данных остается недостаточным. Согласно прогнозу аналитической компании Gartner [1], в период с 2011 по 2016 гг. финансовый ущерб от киберпреступлений ежегодно будет увеличиваться. Корпоративная информация составляет только часть накапливаемых в мире данных. Повсеместное распространение Интернета привело к удвоению объема информации за период 2012–2013 гг. Объем сгенерированных данных в 2012 г. оценивается в 2,8 зеттабайта и прогнозируется до 40 зеттабайт к 2020 г. На сегодняшний день только в России [2] накоплено 155 экзабайт или 2,4 % мировых данных. И в ближайшие семь лет эта доля сохранится. При этом эксперты IDC (International Data Corporation) считают, что сегодняшних хранилищ хватит лишь для 15 % данных [3]. Однако это приемлемо, так как большая часть данных используется краткосрочно и не требует длительного хранения.

Прогнозные исследования в 2012 г. показывают, что объемы информации будут удваиваться каждые два года в течение следующих восьми лет и к 2020 г. их объем должен увеличиться в 15 раз. Одним из основных факторов этого роста является увеличение доли автоматически генерируемых данных: с 11 % от общего объема в 2005 г. до более 40 % в 2020 г. Большие объемы полезных данных создаются с систем видеонаблюдения, встроенных в оборудование, медицинских систем, информации с компьютеров, смартфонов, бытовой электроники. По оценкам IDC, количество устройств в мире, которые можно подключить к Интернету, приближается к 200 млрд, из которых 14 млрд, или 7 %, уже подключены и активно передают данные. На сегодняшний день данные от таких устройств составляют 2 % от мирового объема информации. Согласно прогнозам IDC, к 2020 г. уже 32 млрд подключенных устройств будут генерировать 10 % общего объема данных во всем мире. Объем информации об отдельно взятом пользователе, хранящейся

в цифровой вселенной, станет существенно больше, чем объем данных, создаваемых этим пользователем. Причем большая часть накапливаемой информации плохо защищена. В 2010 г. в защите нуждалось менее трети информации, а к 2020 г. доля такой информации может превысить 40 %.

По прогнозам [4], инвестиции в IT-инфраструктуру цифровой вселенной (оборудование, телекоммуникации, хранение и управление информацией и персонал) в период с 2012 по 2020 г. вырастут на 40 %. Причем инвестиции в хранение и защиту информации, обработку «больших данных» (Big Data) и облачные технологии будут расти значительно быстрее. Большие данные диктуют новые взаимосвязанные принципы обработки информации [5]. Первый — это способность анализировать все данные, а не довольствоваться их частью или статистическими выборками. Второй — готовность иметь дело с неупорядоченными данными в ущерб точности. Третий — изменение образа мыслей: доверять корреляциям, а не гнаться за труднодостижимым поиском причинно-следственных зависимостей.

Существенно и то, что на сегодняшний день используется менее 3 % из 23 % потенциально полезных данных, которые могли бы найти применение с технологиями Big Data.

Беспрецедентный рост информации в мире, необходимость хранения и обработки всей массы накопленных данных требует создания хранилищ, построенных на новых технических средствах, использующих новые модели и методы эффективной обработки данных.

1.2. Недостатки традиционных хранилищ данных

Традиционные системы управления базами данных (СУБД) предназначались для создания и использования информационных моделей — корпоративных баз данных (БД) в конкретных сферах деятельности.

Корпоративные (закрытые) информационные и автоматизированные системы определили условия эксплуатации и требования к их БД:

- предопределенный и ограниченный круг пользователей с фиксированными функциями и правами, а следовательно, относительно определенная и устойчивая структура (схема) данных;
- равномерный рост общего объема данных с малоизменяющимся объемом оперативных данных;
- необходимость независимого совместного доступа (изменения) к данным, обусловившая создание моделей транзакционной обработки БД;
- эффективная работа в реальном времени.

Средства реализации корпоративных информационных систем (КИС), использующие современные серверы баз данных, обеспечивают сформулированные в теореме CAP (теореме Брюера) фундаментальные требования к хранению данных:

- Consistency — согласованность, понимаемая как целостность по ограничениям;
- Avaliability — доступность данных;
- Partition Tollerance — распределение БД по физическим узлам (стабильная работа при линейно растущем объеме).

Наилучшим решением для корпоративной информационной системы оказались многопользовательские централизованные и распределенные базы на основе строго типизированной реляционной модели с транзакционной обработкой данных.

Однако общие тенденции в глобализации производства, электронной коммерции и информатизации общества формулируют новые требования и стимулируют развитие информационных систем:

- создание новых моделей данных, не требующих строго фиксированной структуры;
- использование парадигмы объектно-ориентированного программирования в СУБД;

- расширение круга пользователей с выходом КИС в глобальное информационное пространство с допуском в систему внешних пользователей (поставщиков, потребителей, операторов управления логистикой продукции), работающих с базами данных через WEB-приложения;
- использование содержания запросов и постов в социальных сетях в задачах анализа и прогнозирования деятельности компании.

Новые требования к информационным системам выявили недостатки используемых в них реляционных СУБД:

1. Строгая типизация, приводящая к несоответствию структуры БД структуре данных реального объекта. Для хранения в реляционной базе данные одного информационного объекта должны быть декомпозированы и распределены по множеству равноценных нормализованных таблиц.

2. Атомарность (единственность и неделимость) данных не адекватно представляет множественные свойства и групповые данные.

3. Статичность данных. Серверы реляционных баз данных (РБД) не имеют специальных средств для представления истории изменения данных.

4. Отдельное от информационного объекта хранение и выполнение его собственных действий. Поведение объекта в РБД описывается в виде хранимых в базе функций, процедур и триггеров, не принадлежащих информационному объекту.

5. Плохая масштабируемость, вызывающая стремительное падение производительности при росте объема данных и количества используемых в запросах соединений (JOIN) таблиц.

6. Неустойчивость к отказам оборудования.

При наличии существенных недостатков необходимо помнить и учитывать достоинства реляционной модели данных, обусловливающие ее продолжающееся использование в КИС:

- наглядность исходного табличного представления данных и результатов запросов;

- реляционная полнота языка SQL-запросов, расширенная мощными средствами обработки данных;
- независимость запросов от физической структуры данных (наличия указателей и связей) — возможность построить любой новый запрос без изменений и дополнений в структуре БД.

1.3. Новые источники и области применения хранилищ данных

Развитие функционала в Интернете открыло новые области, требующие хранения и анализа данных:

- массовое размещение и распространение данных и знаний (научно-технических, новостных, экономических, транспортных);
- электронная коммерция — компьютеризированная технология продаж: оповещение, привлечение покупателя, анализ приобретений, программы лояльности к клиенту, направленные на удовлетворение спроса, развитие производства и удержание клиента;
- информационное взаимодействие общества и государства (предоставление госуслуг);
- социальные сети — средство информационного взаимодействия индивидуумов и групп;
- системы связи — БД биллинговых систем операторов связи.

Массивы информации, генерируемой или размещаемой в сети Интернет, предполагают новые задачи и технологии обработки данных:

- новые типы запросов, использующие смысловые отношения, привели к появлению семантического WEBa, основанного на знаниях, размещаемых в сети Интернет;

- исследования интересов общества (анализ содержания и частоты запросов к поисковым системам Интернета);
- анализ содержания сайтов.

Таким образом, новые (постреляционные) хранилища должны сочетать возможности хранения и данных, и знаний в быстро растущих объемах с новыми задачами обработки информации. В целом корпоративные информационные системы, Интернет и системы связи являются движущей силой в области создания новых систем хранения данных.

Новые области применения выдвигают и новые требования к хранилищам данных:

- не атомарность (множественность) и разнородность отдельных атрибутов хранимых объектов;
- разнообразие (не типизируемость) наборов и структур данных хранимых объектов;
- необходимость целостного представления разнородной, как декларативной, так и процедурной информации требует хранения в объекте базы не только данных, но и способов их обработки;
- нелинейный (взрывной) рост объемов хранимых данных.

Появление новых требований к объемам, составу и структуре данных в сочетании с требованиями отказоустойчивости, масштабируемости и эффективности стимулировало развитие хранилищ данных в направлении совершенствования моделей данных, создания средств распределенного хранения и массово параллельных структур для их обработки. В зависимости от значимости того или иного требования в информационной системе создавались хранилища, наилучшим образом соответствующие поставленной задаче. Современное состояние систем для хранения, доступа и обработки данных характеризуется разнообразием используемых моделей данных, средств для распределенного хранения и обработки во множестве узлов вычислительной сети.

2. Классификация постреляционных хранилищ

Системы управления базами данных прошли длительный путь развития, начиная от первых СУБД для IBM и ЕС ЭВМ, использующих иерархическую модель данных, до современных, характеризующихся разнообразием структур и средств обработки данных, предназначенных для применения в информационных системах разного назначения. Хронология появления СУБД, реализующих различные модели данных, представлена на рис. 2.1.

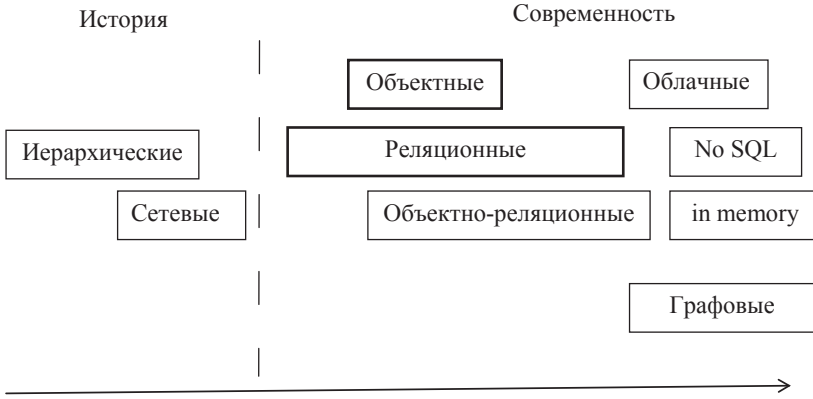


Рис. 2.1. Хронология развития систем управления БД

Под моделью данных обычно понимается система правил, определяющих допустимые структуры для данных и соответствующие им правила поиска и обработки данных. На основе

правил для определения структуры данных создаются языки описания данных (DDL — Data Description Language), а правила поиска и обработки воплощаются в языках манипулирования данными (DML — Data Manipulation Language). Используемая модель данных служит основным признаком для классификации современных хранилищ. Классификация современных хранилищ на основе модели данных приведена на рис. 2.2.

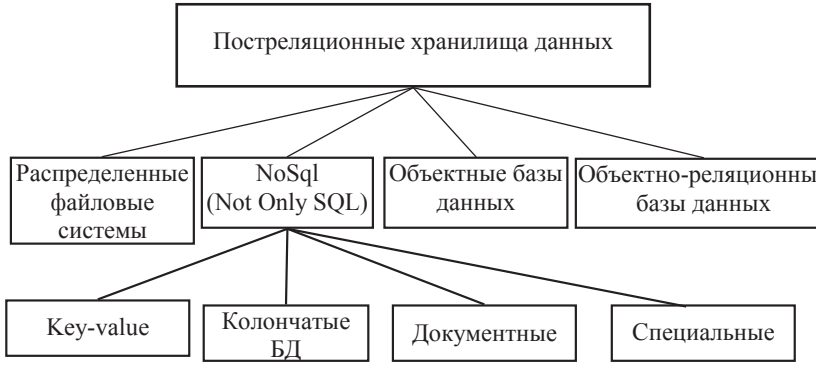


Рис. 2.2. Классификация хранилищ на основе модели данных

Необходимость совместного выполнения противоречивых требований масштабируемости, эффективности и отказоустойчивости обусловила создание и использование средств для распределенного хранения и обработки данных параллельно во многих узлах вычислительной сети. Поэтому естественным решением, позволившим преодолеть ограниченность объема памяти, вычислительной мощности и надежности одного компьютера, являются распределенные хранилища и, в частности, распределенные файловые системы. Они не только решают задачу хранения растущего объема данных на типовых компьютерах, но и являются основой для создания новых СУБД, в том числе и в облачных технологиях.

Стремление сблизить представления данных в базе и обрабатывающей программе на объектно-ориентированном языке привело к появлению объектно-ориентированных баз данных (ООБД). По существу, объектные базы обеспечивают создание стабильных — долговременно хранящихся — программных объектов и средства для поиска и управления этими объектами. Большинство современных ООБД реализуют модель данных, предложенную ODMG 3.0 — Object Data Management Group, которая де-факто считается стандартом в области объектных баз [6]. Ввиду единства парадигмы программирования и хранения данных, доступ и обработка объектных баз выполняется расширенными средствами из языков программирования. Таким образом, объектные БД создавались изначально ориентированными на квалифицированных программирующих пользователей.

В этой классификации объектно-реляционные СУБД рассматриваются как новый этап в развитии SQL-серверов реляционных баз, достигнутый путем наращивания модели новыми, имеющими собственную внутреннюю структуру элементами — коллекциями и объектами.

Особую группу составляют NoSql (Not Only SQL) хранилища данных [7]. В эту группу включаются хранилища, которые используют новые, как табличные, так и не табличные, структуры данных, язык запросов которых не является SQL. Отказ от SQL продиктован необходимостью повысить эффективность запросов за счет исключения длинных цепочек соединений таблиц или переходом к другим (не табличным) структурам данных. Хотя NoSql-хранилища не заменяют SQL-серверы в КИС, для глобальных информационных систем и Интернета эти решения оказываются более эффективны.

Поскольку способ размещения данных в устройствах хранения, тип и место расположения устройств во многом определяют доступность и эффективность хранилищ, эти параметры также используются в качестве признака их классификации:

- хранилища In memory: информационные объекты постоянно хранятся в оперативной памяти серверов, образующих кластер;
- централизованные или распределенные БД — находящиеся в одном устройстве или на множестве взаимодействующих устройств;
- облачные хранилища данных, размещаемые в датацентрах IT-компаний.

Далее рассматриваются способы построения и использования разных типов хранилищ.

3. Объектно-ориентированные СУБД

Объектно-ориентированные СУБД, основываясь на парадигме объектно-ориентированного программирования, должны обеспечивать:

- надежность хранения и эффективность обработки больших объемов данных (большого числа хранимых в базе классов и их экземпляров);
- транзакционность обработки и эффективное управление параллелизмом запросов и внешней памятью;
- развитыми средствами создания запросов, требующих создания языка, сопоставимого по «выразительной мощности» с SQL.

Для стандартизации требований, формализации языков и создания общих архитектурных решений для объектно-ориентированных СУБД была организована международная группа Object Database Management Group (ODMG). Этой группой специалистов был предложен стандарт ODMG-3, ставший ориентиром для разработчиков объектно-ориентированных СУБД.

Стандарт ODMG-3 содержит требования и рекомендации по реализации:

- объектной модели БД Object Data Model (ODM);
- языка определения объектов Object Definition Language (ODL);
- языка запросов к объектной базе Object Query Language (OQL);
- интерфейсов для языков программирования (C++, Java и др.) приложений для объектно-ориентированных БД.

Состав и взаимодействие элементов объектно-ориентированной СУБД показаны на рис. 3.1.

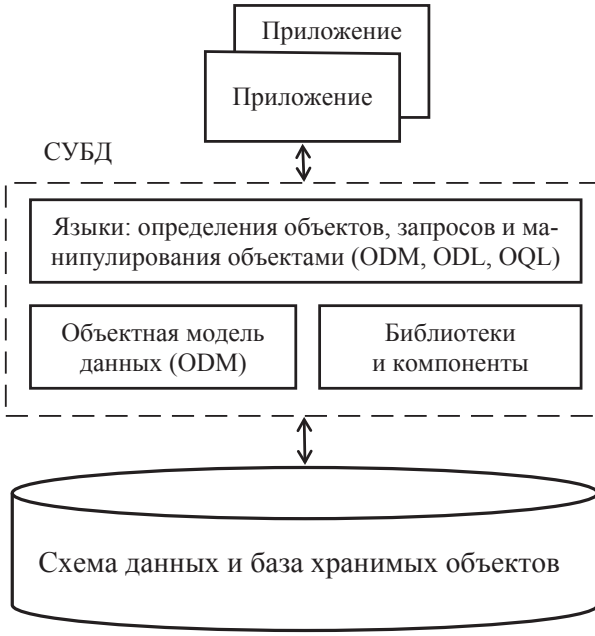


Рис. 3.1. Состав объектно-ориентированной СУБД

Язык определения данных (ODL) — декларативный язык для задания схемы базы данных. Содержит средства для описания схемы данных в виде набора интерфейсов объектных типов (аналог классов):

- описание свойств типов объектов,
- взаимосвязей между типами,
- имен операций (методов для типа) и их параметров.

В стандарте ODMG определяется спецификация языка и конкретизируются особенности объектной модели данных. Стандарт не требует отдельной реализации языка определения данных в СУБД объектных хранилищ. Он может быть реализо-

ван в средах разработки приложений для ООБД. При таком решении определения классов из среды программирования переносятся в схему БД.

Язык манипулирования объектами (OML) — объектно-ориентированный язык программирования отдельных операций и приложений для работы с объектами БД. Языки манипулирования объектами создаются путем интегрирования в язык объектно-ориентированного программирования дополнительных конструкций или библиотек, связывающих программную модель объектов с моделью ODMG. Разработаны спецификации правил связывания, с помощью которых программы на языках C++, Java, Smalltalk и др. могут обращаться к хранимым в базе данным. Схема связывания программных и базовых объектов показана на рис. 3.2.

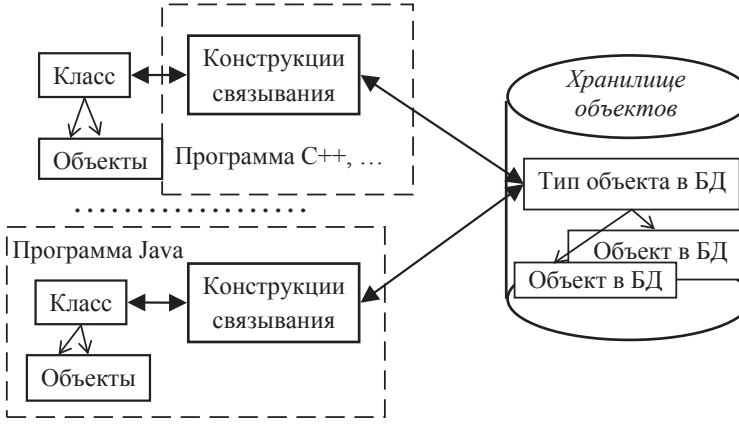


Рис. 3.2. Связывание программных объектов с объектами БД

Язык объектных запросов (OQL**)**. Язык имеет синтаксис, похожий на синтаксис языка SQL, но опирается на семантику объектной модели ODMG (тип, объекты, свойства, методы). В стандарте допускается как прямое использование OQL, так и его встраивание в OML.

Хранилище объектов. Организация объектного хранилища, реализуя на логическом уровне модель ODMG, должна на физическом уровне обеспечивать создание, долговременное хранение, доступ и управление как типами, так и их экземплярами — объектами. В объектном хранилище совместно размещаются структурированные данные и программные коды. Стандарт оставляет за разработчиками ООБД физическую организацию размещения иерархии типов и объектов в долговременной памяти.

Средства разработки ООБД и приложений содержат:

- интерактивные средства и трансляторы с языков ODL и OML для создания иерархии объектных типов (схемы) и их объектов в БД;
- иерархии классов для создания графических интерфейсов отображения данных в диалоговых формах;
- встроенные в предопределенные классы обработки литералов (строк, чисел, использование ссылок).

Состав объектно-ориентированной модели ODMG-3

ODMG является композиционной объектной моделью, включающей возможность описания в БД как объектов, так и простых литеральных значений.

В концепции ODMG объектная специфика данных реализуется следующим образом [8].

1. Модель одновременно включает понятия объектов и литералов. Каждый объект имеет уникальный идентификатор и наборы изменяемых свойств. Литерал не имеет идентификатора и представляет константу в различных структурах.

2. Модель содержит исходный набор встроенных объектных и структурных типов (связей), позволяющих создавать новые типы и моделировать схемы данных.

3. Объекты — экземпляры определенного типа (рис. 3.3). Все объекты одного типа имеют одинаковое множество свойств и одинаковое поведение (множество операций), определяемое в типе.

4. База данных состоит из наборов объектов, относящихся к описанным в схеме типам.

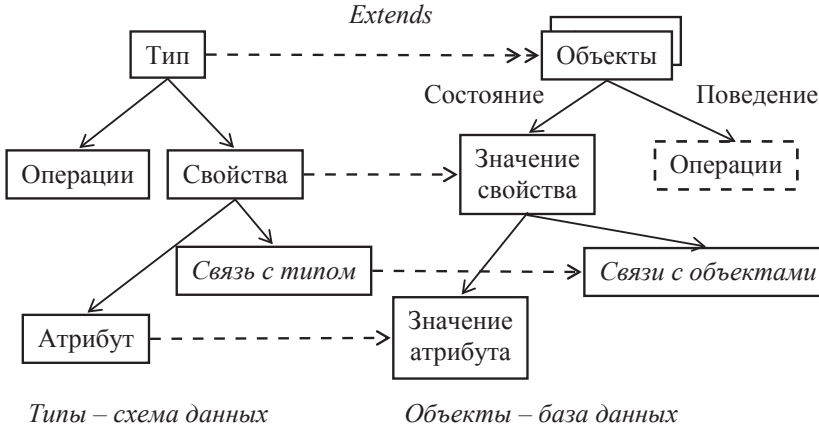


Рис. 3.3. Соответствие понятий ООБД

При описании объектного типа на языке ODL определяется его интерфейс и реализация на включающем языке. Интерфейс объектного типа содержит:

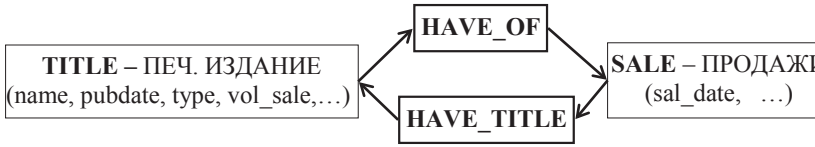
- имя типа;
- исходный тип (супертип), а при множественном наследовании — набор используемых исходных типов;
- набор свойств — атрибутов: свойство может быть другим типом или литералом, свойство литерального типа может быть коллекцией;
- набор свойств — именованных связей: каждая связь указывает на другой объектный тип, связи поддерживаются между объектами в заданной кратности «1 : 1», «1 : M», «M : N» и определяются как двухсторонние, СУБД контролирует правильность связи со стороны каждого объекта;

- набор операций (методов или функций) с указанием их имен и параметров: реализация операции описывается средствами включающего языка (C++, Java,..).

Структура описания типа:

Типы-предки		Свойства-типы		Свойства-литералы			Связи		Операции
Имена типов	...	Имена типов	...	Значение	Коллекция	...	Имя связи, имя типа	...	Имя и параметры

Рассмотрим пример определения схемы данных, содержащей два типа — **TITLE** (ПЕЧАТНОЕ ИЗДАНИЕ) и **SALE (**ПРОДАЖИ) — и связи между ними:




```
class TITLE
{name string [120], pubdate Date, type string
[80], …
relationship SALE HAVE_OF inverse SALE:: HAVE_
TITLE …
vol_sale ()}
```

В типе `TITLE` описываются свойства `name, pubdate` и др. и связь `HAVE_OF` от объекта типа `TITLE` к объектам типа `SALE`. Кроме того, в типе `TITLE` определяется операция `vol_sale`. Аналогично описывается тип `SALE`, в котором определена обратная связь `HAVE_TITLE` от объекта типа `SALE` к объектам типа `TITLE`.

```
class SALE
{sal_date Date, …
relationship TITLE HAVE_TITLE inverse TITLE::
HAVE_OF, …}
```

На заданной схеме предложениями OQL строятся запросы к ООБД. Реализацией OQL является язык SQL3, представляющий расширение SQL 92. Основу SQL3 составляет оператор SELECT, который может возвращать набор литеральных значений, являющийся объектами БД, свойствами объектов и результатами операций.

Рассмотрим примеры запросов в OQL на вышеприведенной схеме печатных изданий и их продаж. Пусть в БД множество объектов (экстент) типа `TITLE` называется `TITLES`, а экстент типа `SALE – SALES`.

1. Построим запрос на получение свойств `name и pubdate` объектов типа `TITLE`, удовлетворяющих условиям на значение свойства `type`:

```
SELECT DISTINCT STRUCT (Name: T.name, Pdate:
T.pubdate)
FROM TITLES T WHERE T.type = 'business';
```

Форма запроса совпадает с SELECT- SQL, но такой запрос вернет литерал — массив, элементами которого являются значения типа `struct {string [80] Name; date Pdate}` — структуры с полями `Name` (название) и `Pdate` (дата издания), выбранные из объектов книг по бизнесу `(type = 'business')`.

2. Следующий запрос демонстрирует использование операций. Если при описании типа ПЕЧАТНОЕ ИЗДАНИЕ (`TITLE`) определена операция `vol_sale`, вычисляющая общую сумму, вырученную от продажи издания, используя данные в связанных объектах `SALES`, то запрос:

SELECT DISTINCT STRUCT (Name: T. TITLE, Summa: T.vol_sale)

FROM TITLES T WHERE T. TYPE = 'business';

вернет массив структур, элементами которых являются названия и суммы выручки для каждой книги по бизнесу.

3. Рассмотрим запрос с использованием связи HAVE_OF, установленной между объектными типами TITLE и SALE.


```
  SELECT DISTINCT
  STRUCT (Name: T.name, SALE: (SELECT S FROM T.
HAVE_OF AS S
  WHERE S.sal_date > '01/01/2015'))
  FROM TITLES T;
```

Результат запроса — множество {Name <*name*> {SALE <*S*>, …}, …}, элементами которого являются структуры с двумя значениями:

- первое — атомарное литеральное значение <*name*> — название книги (типа STRING),
- второе — множество {SALE <*S*>, ...} с элементами-объектами типа SALE — ПРОДАЖИ, связанными по связи HAVE_OF с экземплярами типа ПЕЧАТНОЕ ИЗДАНИЕ и датой продаж больше ‘01/01/2015’.

4. Объектно-ориентированные СУБД поддерживают уникальные имена — идентификаторы объектов (OID — Object Identifier), которые допускают прямое обращение к объекту БД по его OID.

Например, если в БД есть издание с идентификатором *BU1111*, то результатом обращения *BU1111* будет литерал, соответствующий объекту.

Результатом запроса *BU1111.HAVE_OF* будет множество, состоящее из объектов типа ПРОДАЖИ, полученных по связи *HAVE_OF* с объектом *BU1111*.

Рекомендации ODMG-3 учитывались в разработке многих коммерческих ООБД (Cache, Poet, Jasmine). Однако практика использования ООБД выявила недостатки стандарта ODMG 3.0:

- объектная модель весьма сложна и при этом недостаточно обоснована;

- язык OQL является простым с концептуальной точки зрения, но чрезмерно сложен для использования непрограммистами;
- зависимость запросов от структуры и связей типов в схеме базы данных вызывает трудности эффективной реализации новых, не предусмотренных на этапе создания БД запросов.

По этим причинам ООБД использовались преимущественно в специализированных информационных системах.

4. Объектно-реляционные БД

Ограничение классической (по Кодду) модели данных только нормализованными таблицами не позволяет адекватно (в рамках одной таблицы) представлять информационные объекты, имеющие сложную внутреннюю структуру. Объектно-реляционная модель расширяет реляционный подход системой новых структурных типов данных. Среди новых типов появились как специальные (геометрические, географические), так и универсальные типы (массивы, коллекции, объекты). Рассмотрим коллекции и объекты как наиболее адекватные структуры для построения информационных моделей. Наиболее полно и последовательно концепции коллекций и объектных типов реализованы в СУБД Oracle [9].

4.1. Коллекции в базе Oracle

Коллекцией называют упорядоченную группу элементов одного типа. Единицей доступа является элемент коллекции. Oracle поддерживает несколько видов коллекций [10]. Языки SQL и PL/SQL в Oracle содержат средства для их создания и работы с ними. Три типа коллекций отличаются способами хранения и доступа к элементам.

1. **Вложенные таблицы (nested tables)** — упорядоченный набор однотипных элементов, который «привязан» к полю (клет-

ке) таблицы. Вложенными таблицами удобно представлять в БД объекты с множественными атрибутами. Для столбца, имеющего тип коллекции, создается вспомогательная («вложенная») таблица. Элементы коллекций из полей главной таблицы хранятся в столбце вложенной таблицы. Между строками главной и вложенной таблиц автоматически создается и поддерживается связь «1 : M». Средства SQL и PL/SQL обеспечивают работу с коллекцией как в виде списка элементов в поле главной таблицы, так и виде значений столбца связанной таблицы.

2. **Ассоциативные массивы (associative arrays)** — одномерные индексированные наборы однородных элементов, доступные только в PL/SQL. Значение элемента может быть ассоциировано с числом и текстовой строкой.

3. **Массивы переменной длины (variable-size arrays)** — ограниченные наборы однородных элементов, доступные в БД и PL/SQL.

Вложенные таблицы (ВТ) наиболее удобное средство расширения реляционной модели данных.

Создание и обработка коллекции «вложенная таблица» средствами SQL

Использование коллекции в виде вложенной таблицы требует предварительного создания специального типа данных — «вложенная таблица». В команде создания определяется имя для типа ВТ и задается тип для элементов коллекции. Тип, представляющий вложенную таблицу, сохраняется в БД.

Для создания типа используется SQL-команда:

```
CREATE Type <Тип ВТ> IS Table OF <Тип элементов ВТ>;
```

Сохраненный в базе тип ВТ далее может использоваться в качестве типа для столбца при создании пользовательской таблицы, содержащей коллекции. Коллекции «хранятся» в полях этого столбца:

```
CREATE TABLE < Имя таблицы > (
<Описание обычного столбца>, …
<Имя столбца, содержащего коллекции> <Тип ВТ>, …)
NESTED TABLE <Имя столбца, содержащего коллекции>
STORE AS <Имя таблицы БД для хранения коллекций>;
```

Добавление в таблицу строки с полем, содержащим коллекцию, выполняет обычная команда INSERT, в которой значение коллекции задается конструктором, именем которого служит имя типа для столбца, а параметрами — элементы загружаемой коллекции:

```
INSERT INTO <Имя таблицы> (…,<Имя столбца-коллекции>,..)
VALUES (…,<Имя типа ВТ> (<Элемент коллекции 1>,…), …);
```

Рассмотрим пример создания и обработки коллекции. Пусть в таблице авторов адрес автора решено задать коллекцией, чтобы иметь возможность указать несколько адресов (домашний, рабочий, электронной почты и др.). Каждый адрес имеет строковый тип varchar2 (50). Сначала создается тип вложенной таблицы, который будет использоваться в столбце при создании и заполнении таблицы AUTHORS_C.

— Создание типа `Adr_Col_t` *коллекции — вложенная таблица:*

```
CREATE Type Adr_Col_t IS Table OF varchar2 (50);
```

— Создание таблицы авторов AUTHORS_C *с коллекцией в столбце ADR_COL.*

```
CREATE TABLE "SYSTEM"."AUTHORS_C"
("AU_ID" VARCHAR2 (11 BYTE),— идентификатор автора
"AU_LNAME" VARCHAR2 (40 BYTE),— фамилия автора
"AU_FNAME" VARCHAR2 (20 BYTE),— имя автора
"CONTRACT" NUMBER (1,0),— наличие контракта
```

```
  "ADR_COL" Adr_Col_t — адреса автора
  ) NESTED TABLE "ADR_COL" STORE AS In_Tab_ADR;
  — Добавление в таблицу нового автора
  INSERT INTO AUTHORS_C VALUES ('000-00-0001',
'Бунин', 'Иван', 1, Adr_Col_t ('Екатеринбург,
Гл.проспект 1',
  Москва, Тверская 2',));
```

Запрос к таблице, содержащей коллекцию:

```
SELECT * FROM < Имя таблицы>;
```

выводит в поле, содержащем коллекцию, список ее элементов.

Запрос коллекции из конкретного поля:

```
SELECT <Имя столбца с коллекцией> FROM <Имя
таблицы> WHERE <Условие для выбора из таблицы
одной строки>;
```

вернет коллекцию в форме списка элементов.

Преобразование коллекции из списка элементов в таблицу выполняет функция `TABLE (<коллекция>)`. Функция `TABLE (…)` представляет элементы коллекции в виде таблицы с единственным столбцом с именем `COLUMN_VALUE`. Таблица, созданная функцией `TABLE (…)`, доступна по команде `SELECT`:

```
SELECT COLUMN_VALUE AS <Новое имя столбца>
FROM TABLE (<Запрос, возвращающий из таблицы
поле, содержащее коллекцию>);
```

Преобразование коллекции в одностолбцовую таблицу предоставляет возможность соединения элементов коллекций с содержащими их строками:

```
SELECT Master.*, Nest.*
FROM <Главная таблица> Master, TABLE
(Master.<Поле-
коллекция>) Nest
[WHERE … и др. параметры запроса];
```

Здесь и далее в описании синтаксиса команд символы `[…]` обозначают необязательность параметра.

Master и Nest — примеры псевдонимов для главной и вложенной таблиц. Запрос вернет таблицу, в которой каждая строка главной таблицы повторяется с одним элементом коллекции, хранящейся в поле этой строки. Параметром WHERE могут быть отфильтрованы строки главной таблицы по условию на элементы их коллекций.

Функция CARDINALITY (<*Коллекция*>) возвращает число элементов в коллекции.

Добавление в коллекцию нового элемента:

```
INSERT INTO TABLE (<Select поля, содержащего коллекцию>) VALUES (<Добавляемый элемент>);
```

Замена элемента в коллекции:

```
UPDATE TABLE (<Select поля, содержащего коллекцию>)
SET COLUMN_VALUE = <Новое значение элемента>
WHERE <Условие для поиска в коллекции заменяемого элемента>;
```

Например, обновление элемента в коллекции адресов (ADR_COL) для автора с AU_ID = '000–00–0001' выполнит команда:

```
UPDATE TABLE (SELECT ADR_COL FROM AUTHORS_C
WHERE AU_ID = '000-00-0001')
SET COLUMN_VALUE = 'Париж, Монмартр, 2';
WHERE COLUMN_VALUE Like '% Москва%';
```

Вывод измененной коллекции для проверки:

```
Select ADR_COL From AUTHORS_C
WHERE AU_ID = '000-00-0001';
```

Удаление элемента из коллекции:

```
DELETE FROM TABLE (<Select поля, содержащего коллекцию>
WHERE <Условие для поиска в коллекции удаляемого элемента>;
```

Например,

```
DELETE FROM TABLE (SELECT ADR_COL FROM AUTHORS_C
                   WHERE AU_ID = '000-00-0001')
```

```
WHERE COLUMN_VALUE Like '%Екатеринбург%';
```

Кроме функции TABLE (…) доступ к вложенной таблице предоставляет функция THE (<*Коллекция*>). Однако функция THE (…) не позволят соединять в запросе строки главной и вложенной таблиц.

Создание и обработка коллекций в программе PL\SQL

Для чтения коллекций из базы в программе объявляется:

- объект-коллекция, имеющий тот же тип, что и коллекция в базе;
- переменная, имеющая тип элемента коллекции.

— Объявление объекта-коллекции в программе для чтения из БД:

```
DECLARE <Имя объекта-коллекции в программе>
<Тип ВТ>;
<Имя переменной для элемента коллекции > <Тип>;
J integer:= 0;
— Например:
AUTHOR_ADR — Имя объекта-коллекции (адреса автора)
ADR1 — Элемент коллекции (один адрес)
BEGIN
— Чтение коллекции из базы:
SELECT <Поле-коллекция в главной таблице>
INTO <Имя объекта-коллекции> FROM <Имя главной таблицы>
WHERE <Условие для выбора одной строки в гл. таблице>;
— Последовательный доступ к элементам коллекции через индекс:
FOR J IN <Имя коллекции>.FIRST.. <Имя коллекции>.LAST
LOOP
```

```
DBMS_OUTPUT.PUT_LINE (<Имя коллекции> (J)); — Вывод
J-го элемента
END LOOP;
```

Например, получение в переменной программы AUTHOR_ADR коллекции адресов автора с AU_ID = '000-00-0001':

```
Select ADR_COL INTO AUTHOR_ADR From AUTHORS_C
WHERE AU_ID = '000-00-0001';
```

Вывод адресов из прочитанной коллекции:

```
FOR J IN AUTHOR_ADR. FIRST.. AUTHOR_ADR. LAST
LOOP
DBMS_OUTPUT. PUT_LINE (AUTHOR_ADR (J)); — Вывод J-го эл-та
END LOOP;
```

— Добавление в таблицу БД строки, в которой значение коллекции берется из программы:

```
INSERT INTO <Имя главной таблицы> (…,<Поле-коллекция>, …)
VALUES (…, <Имя коллекции>, …);
```

Например, добавление автора с коллекцией адресов из AUTHOR_ADR:

```
INSERT INTO AUTHORS_C VALUES ('000-00-0002', 'Дюма,
'Александр', 0, AUTHOR_ADR);
```

Методы объекта-коллекции в программе

1. Получение количества элементов в коллекции:

```
<Объект-коллекция>.COUNT
```

— Например, вывод количества элементов в коллекции:

```
DBMS_OUTPUT.PUT_LINE (AUTHOR_ADR. COUNT);
```

2. Доступ к первому/последнему элементу в коллекции:

```
<Объект-коллекция>.FIRST/LAST
```

— Например, вывод последнего элемента из коллекции:

```
DBMS_OUTPUT. PUT_LINE (AUTHOR_ADR. LAST);
```

3. Присваивание значения элементу коллекции, с доступом по индексу элемента:

<Объект-коллекция> (<Числовой индекс>):=<Выражение, имеющее тип элемента коллекции>;

4. Добавление нового элемента в объект-коллекцию — метод EXTEND.

EXTEND без параметров добавляет один элемент с признаком NULL,

EXTEND (n) добавляет n элементов NULL,

EXTEND (n, i)) добавляет n элементов со значением i-го элемента.

Например, AUTHOR_ADR. EXTEND (3, 1);

5. Удаление элемента из коллекции — метод DELETE.

DELETE без параметров удаляет все элементы из коллекции,

DELETE (i) удаляет i-ый элемент из коллекции,

DELETE (i, j) удаляет из коллекции элементы с i-го по j-ый.

При удалении промежуточных элементов сжатие коллекции не выполняется. В позициях удаленных элементов значения не определены.

6. Проверка наличия в коллекции элемента с заданным индексом:

<Объект-коллекция>.EXISTS (*<Индекс>*); — возвращает значение TRUE/FALSE.

Пример чтения коллекции и проверки ее элементов:

```
Select ADR_COL INTO AUTHOR_ADR From AUTHORS_C
WHERE AU_ID = '000-00-0002';
```

— Проверка наличия элементов в коллекции

```
FOR J IN AUTHOR_ADR.FIRST.. AUTHOR_ADR. LAST
LOOP
IF AUTHOR_ADR. EXISTS (J)
Then
DBMS_OUTPUT. PUT_LINE ('Есть элемент под №' || J);
```

```
  Else
  DBMS_OUTPUT. PUT_LINE ('Нет элемента под №'
|| J);
  End if;
  END LOOP;
```

Коллекции — удобное средство для хранения информационных объектов, имеющих множества однотипных данных. Однако коллекции не позволяют иметь элементы разных типов и описывать «поведение» объекта. Для этой цели в Oracle введены объектные типы. Так как объектные типы создаются в базе, они могут использоваться для определения элементов коллекций. Таким образом, в таблицах Oracle могут храниться коллекции из объектов.

4.2. Объекты в БД Oracle

Объект Oracle — хранимый экземпляр особого создаваемого в БД типа данных. Объекты объединяют в единую программную или хранимую в базе структуру атрибуты (свойства) и методы (действия) реальных объектов [11]. Объекты размещаются в столбце таблицы, имеющем созданный пользователем объектный тип. Средства Oracle позволяют читать объекты из базы в PL\SQL-программу, обрабатывать их и сохранять в БД и (рис. 4.1).

Для размещения объектов в таблицах БД существует два способа (рис. 4.2). В первом объект занимает всю строку, поэтому в таблице не может быть других данных. Вся таблица имеет объектный тип и называется таблицей со строками-объектами. В SQL обращение к свойствам и методам объекта, хранящегося в строке, осуществляется только через алиас таблицы: *<алиас таблицы>.<имя свойства/имя метода>*.

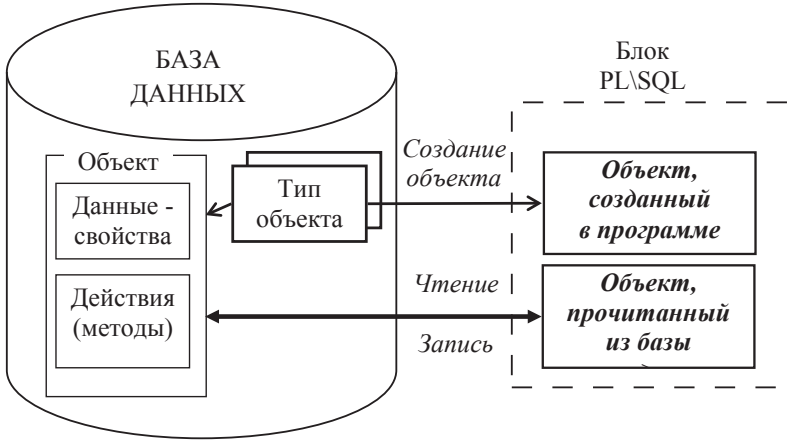


Рис. 4.1. Создание и использование объектов в программе

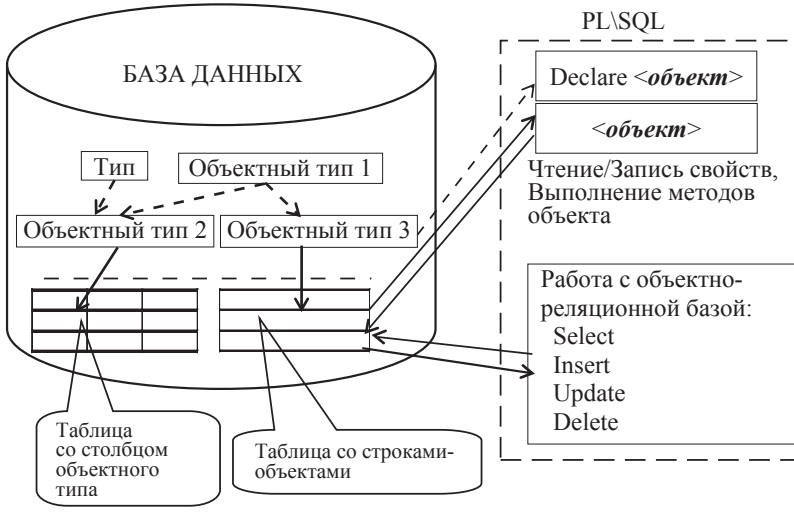


Рис. 4.2. Способы размещения объектов в базе

Другой способ хранения объекта — размещение объекта в поле таблицы. В этом случае объектный тип используется при определении отдельного столбца, а объекты хранятся в полях

этого столбца. В SQL обращение к свойствам и методам такого объекта осуществляется через алиас таблицы и имя столбца:

<алиас таблицы>.<имя столбца-объекта>.<имя свойства/имя метода>.

4.2.1. Объектный тип Oracle

Определение объектного типа Oracle состоит:

- из обязательной спецификации (спецификация свойств и методов для типа);
- необязательного тела (кода программ для методов).

При построении системы типов разрешено использовать наследование.

Создание спецификации объектного типа

```
CREATE [OR REPLACE] TYPE <Имя типа>
[AUTHID {CURRENT USER | DEFINER
AS OBJECT
(<Имя свойства> <Тип данных>,— определение свойства
..........
[<Спецификация метода>,— заголовки процедур и функций
..........]);
```

Опция `[AUTHID {CURRENT USER | DEFINER}]` определяет права, которые проверяются при выполнении методов: текущего или владельца схемы.

Тип данных для свойства может быть любым, используемым в SQL, кроме LONG, LONG RAW, ROWID, UROWID. Типом данных свойства также может быть другой объектный тип.

Методы могут быть четырех видов, имеющих следующие спецификации:

— методы-члены, выполняющие операции для объектов:

MEMBER {PROCEDURE | FUNCTION} < *Спецификация программы*>;

— статические методы, выполняющиеся для типа и не требующие создания объекта:

STATIC {PROCEDURE | FUNCTION} < *Спецификация программы*>;

— методы-конструкторы, выполняющиеся при создании объектов:

CONSTRUCTOR FUNCTION < *Спецификация функции* >;

— методы сравнения, реализующие функцию, которая проверяет, являются ли два объекта равными:

{ORDER|MAP} MEMBER FUNCTION < *Спецификация функции сравнения* >.

Создание тела для объектного типа

Тело объектного типа содержит программы методов, заданных в спецификации типа:

```
CREATE [OR REPLACE] TYPE BODY < Имя типа >
AS
{MEMBER|STATIC}{PROCEDURE|FUNCTION} <Тело метода>;
......
[MAP MEMBER FUNCTION <Тело функции сравнения>;]
END;
```

Использование объектных типов

Создание таблицы со столбцами объектного типа

```
CREATE TABLE <Имя таблицы>
(<Имя обычного столбца> <Тип столбца>, …,
< Имя столбца-объекта> <Имя объектного типа>, …);
```

При работе в SQL доступ к свойству или методу объекта при вложенности объектных типов задается иерархическим путем:

<имя столбца>.<имя свойства-типа >. ....<имя свойства/метода>.

4.2.2. Создание и обработка объектов в БД

Применение для объектов нетрудно найти в любой базе. Для примера используем базу издательства. Сведения об авторе представим в виде объекта с типом `CitizenType` (*тип для гражданина*), имеющим свойства:

```
au_id varchar2 (11), — идентификатор автора
au_lname varchar2 (40), — фамилия автора
au_fname varchar2 (20), — имя автора
phone char (6), — номер телефона автора
birth_date date, — дата рождения автора
address: address_type — адрес
    postal_code VARCHAR2 (6) — почтовый индекс
    state VARCHAR2 (12), — страна
    city VARCHAR2 (20), — город
    street VARCHAR2 (30), — улица
    numb NUMBER — номер дома
```

Свойство `address` содержит несколько элементов и определяется своим типом `address_type`. Тип для адреса не имеет методов.

Тип `CitizenType` имеет два метода-члена:

— метод для вычисления возраста автора (в годах) по дате его рождения;

— метод для вывода фамилии, имени и адреса автора.

Сначала необходимо создать тип для адресов авторов – `address_type`. Так как этот тип не имеет методов, то для него достаточно задать только спецификацию:

```
CREATE TYPE address_type AS OBJECT
(postal_code VARCHAR2 (6), — почтовый индекс
state VARCHAR2 (12),— страна
city VARCHAR2 (20),— город
street VARCHAR2 (30),— улица
numb NUMBER); - номер дома
```

Теперь можно создать тип `CitizenType` для авторов.

Спецификация для типа `CitizenType`:

```
CREATE OR REPLACE TYPE CitizenType AS OBJECT
(au_id varchar2 (11),
au_lname varchar2 (40),
au_fname varchar2 (20),
phone char (6),
birth_date date,
address address_type,— использование типа для адреса
MEMBER FUNCTION age RETURN NUMBER,— метод вычисления возраста
MEMBER PROCEDURE print_address (SELF IN
CitizenType));
```

Параметр `SELF` в спецификации метода `print_address` обеспечивает в программе метода доступ к самому объекту.

Создание тела (определение методов) для типа `CitizenType`

```
CREATE OR REPLACE TYPE BODY CitizenType
AS
— метод для вычисления возраста
MEMBER FUNCTION age RETURN NUMBER
IS
BEGIN
RETURN EXTRACT (YEAR FROM SYSDATE) —
EXTRACT (YEAR FROM SELF.birth_date);
END;
— Метод для вывода сведений об авторе
MEMBER PROCEDURE print_address (SELF IN
CitizenType) IS
BEGIN
DBMS_OUTPUT. PUT_LINE (au_lname || ' '|| au_
fname);
DBMS_OUTPUT. PUT_LINE (address.postal_code||',
'||address.state||', '||address.city);
DBMS_OUTPUT. PUT_LINE (address.street ||', '||
```

```
      TO_CHAR (address.numb));
  END;
  END;
```

1. Создание таблицы со столбцом AUTHOR для объектов типа CitizenType

```
  CREATE TABLE authors_1 (ID_AUTH INT,
NICK VARCHAR2 (12),
  AUTHOR CitizenType);
```

Доступ и управление объектами, хранящимися в столбце таблицы БД

При вставке в таблицу строк командой `INSERT` значения для свойств добавляемого объекта задаются параметрами конструктора. Если метод конструктора (CONSTRUCTOR FUNCTION) в теле типа не задан, то используется конструктор по умолчанию, имеющий само имя типа.

```
  INSERT INTO authors_1 VALUES (1, 'Первый',
  CitizenType ('111-11-1111', 'Попов', 'Юрий',
'121212',
  '31.01.1990', address_type ('123456', 'РФ',
'Чита','Мира', 3)));
  INSERT INTO authors_1 VALUES (2, 'Второй',
  CitizenType ('222-22-2222',
'Светов','Иван','232323',
  '30.04.1991', address_type ('133333','РФ',
'Казань','Мира',5)));
```

— Выборка из таблицы добавленных строк:

```
  select nick, AUTHOR From authors_1;
```

	NICK	AUTHOR
1	Первый	SYSTEM.CITIZENTYPE('111-11-1111','Попов','Юрий','121212','1990-01-31 00:00:00.0',SYSTEM.ADDRESS_TYPE('123456','РФ','Чита','Мира',3))
2	Второй	[SYSTEM.CITIZENTYPE]

— Выборка отдельных свойств объекта:

```
  select a.nick, a.AUTHOR.au_lname, a.AUTHOR.
address.City From authors_1 a;
```

Задание алиаса (в примере `a`) для таблицы, содержащей объект, обязательно.

	NICK	AUTHOR.AU_LNAME	AUTHOR.ADDRESS.CITY
1	Первый	Попов	Чита
2	Второй	Светов	Казань

— Замена значения свойства в объекте:

```
update authors_1 a
set a.AUTHOR.address.City='Екатеринбург'
where a.nick = 'Первый';
select nick, AUTHOR From authors_1;
```

	NICK	AUTHOR
1	Первый	SYSTEM.CITIZENTYPE('111-11-1111','Попов','Юрий','121212','1990-01-31 00:00:00.0',SYSTEM.ADDRESS_TYPE('123456','РФ','Екатеринбург','Мира',3))
2	Второй	[SYSTEM.CITIZENTYPE]

— Использование в SQL-запросе методов объекта (вызов метода age):

```
select a.nick as "Ник автора", a.AUTHOR.age ()
as ВОЗРАСТ From authors_1 a;
```

	Ник автора	ВОЗРАСТ
1	Первый	25
2	Второй	24

Обработка объектов в программе PL/SQL

```
DECLARE
```

— Объявление в программе объекта AVTOR типа CitizenType:

```
AVTOR CitizenType;
```

— Объявленный объект AVTOR не определен (NULL):

`VOZRAST NUMBER;` *— объявление числовой переменной* VOZRAST

```
BEGIN
```

— Чтение объекта из поля таблицы в программный объект AVTOR:

```
select a.AUTHOR INTO AVTOR From authors_1 a
where a.nick = 'Первый';
```

— Вывод свойств объекта из программы:

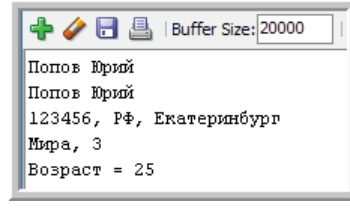
```
DBMS_OUTPUT.PUT_LINE (AVTOR.au_lname||' '||
AVTOR.au_fname);
```

— Выполнение методов программного объекта AVTOR:

```
AVTOR.print_address ();
```
— *выполнение метода-процедуры*

```
VOZRAST:= AVTOR.age ();
```
— *выполнение метода-функции*

```
DBMS_OUTPUT. PUT_LINE ('Возраст = '|| TO_CHAR
(VOZRAST));
END;
```



2. Создание таблицы с размещением объекта в строке

Такая таблица всегда содержит один столбец объектного типа, который не имеет отдельного имени. Для создания таблицы достаточно сослаться на объектный тип.

```
CREATE TABLE < Имя таблицы > OF <Имя объектного типа>;
```

Обращение в SQL к свойству или методу объекта выполняется через псевдоним таблицы: ***<алиас таблицы>.<имя свойства/имя метода>.***

Например, создание таблицы для объектов, имеющих тип `CitizenType`:

```
CREATE TABLE authors_2 OF CitizenType;
```

— Загрузка таблицы объектами:

```
  INSERT INTO authors_2 VALUES
  (CitizenType ('111-11-1111', 'Попов', 'Юрий',
'121212', '31.01.1990',
  address_type ('123456','РФ', 'Чита','Мира',
3)));
  INSERT INTO authors_2 VALUES
  (CitizenType ('222-22-2222',
'Светов','Иван','232323', '30.04.1991',
  address_type ('133333','РФ','Казань','Ми
ра',5)));
```

— Проверка данных в таблице SQL-запросом:

```
select * From authors_2;
```

— выводит все свойства объектов в отдельные столбцы

	AU_ID	AU_LNAME	AU_FNAME	PHONE	BIRTH_DATE	ADDRESS
1	111-11-1111	Попов	Юрий	121212	31.01.90	SYSTEM.ADDRESS_TYPE('123456','РФ','Чита','Мира',3)
2	222-22-2222	Светов	Иван	232323	30.04.91	[SYSTEM.ADDRESS_TYPE]

— Доступ в SQL к отдельным свойствам выполняется через алиас таблицы:

```
  select a2.au_lname, a2.address.City From
authors_2 a2
  where a2.address.State = 'РФ' and
  a2.address.City <> 'Казань';
```

— Выполнение метода объекта в SQL:

```
  select a2.au_lname, a2.age () From authors_2 a2;
```

— В SQL выборка объекта из строки таблицы выполняется функцией

VALUE (*<алиас таблицы>*):

```
  select VALUE (a2) From authors_2 a2
  where a2.au_id = '222-22-2222';
```

	VALUE(A2)
1	SYSTEM.CITIZENTYPE('222-22-2222','Светов','Иван','232323','1950-01-01 00:00:00.0',SYSTEM.ADDRESS_TYPE('133333','РФ','Казань','Мира',5)

Обработка объектов в программе PL/SQL

— Выборка объекта из строки таблицы в PL\SQL также требует использования функции VALUE (<*алиас таблицы*>):

```
DECLARE
AVTOR CitizenType; — объявление объекта в программе, для чтения объектов из таблицы authors_2
birth DATE; — объявление переменной «дата рождения» для свойства birth_date
BEGIN
— Чтение объекта из строки таблицы в программный объект:
select VALUE (a2) INTO AVTOR From authors_2 a2
where a2.au_id = '222-22-2222';
— Выполнение метода программного объекта:
DBMS_OUTPUT. PUT_LINE ('Возраст = '||TO_CHAR (AVTOR.age ()));
— Изменение свойства объекта в программе:
AVTOR.birth_date:= '01.01.1950';
— Изменение свойства объекта в таблице:
UPDATE authors_2 a2 set a2.birth_date = AVTOR. birth_date where a2.au_id = '111-11-1111';
— Замена программного объекта в строке таблицы:
UPDATE authors_2 a2 set VALUE (a2) = AVTOR
where a2.au_id = '222-22-2222';
— Проверка изменений в объектной таблице чтением свойства:
select a2.birth_date INTO birth From authors_2 a2
where a2.au_id = '111-11-1111';
DBMS_OUTPUT. PUT_LINE ('Дата рождения = '||
TO_CHAR (birth, 'dd/mm/yyyy'));
select a2.birth_date INTO birth From authors_2 a2
where a2.au_id = '222-22-2222';
DBMS_OUTPUT. PUT_LINE ('Дата рождения = '||
TO_CHAR (birth, 'dd/mm/yyyy'));
END;
```

4.2.3. Использование ссылок для связывания объектов

В таблицах со строками-объектами каждая строка имеет уникальный идентификатор **Object ID**, который может быть использован для создания ссылки (REF) на этот объект из других таблиц или в тексте программы.

Ссылку на объект, хранящийся в строке таблицы, возвращает функция REF (<*Объект*>). Объект из таблицы по ссылке читает функция DEREF (<*Ссылка* >). Использование функций для работы со ссылками показано на рис. 4.3.

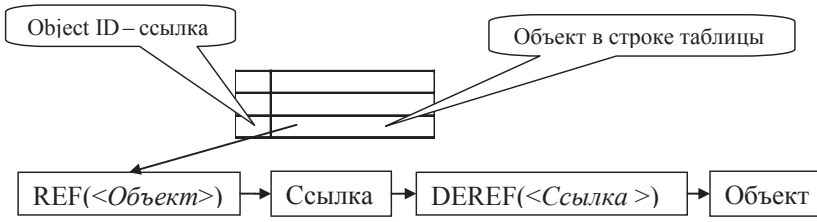


Рис. 4.3. Функции для работы со ссылками на объекты

Ссылки на объекты можно хранить в таблице базы. Для столбца ссылок в указание типа добавляется модификатор REF:

<*Имя столбца*> REF <*Имя объектного типа*>.

Переменная программы, принимающая значение ссылки, также объявляется с модификатором REF: <*Имя переменной*> REF <*Имя объект. типа*>.

Использование ссылок в SQL-программе

Получение ссылки на объект, находящийся строке таблицы:

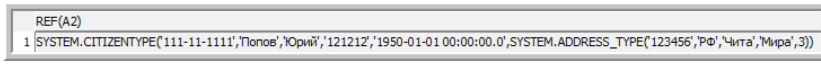
SELECT REF (<*Алиас таблицы*>)

FROM <*Имя таблицы*> <*Алиас таблицы*>

WHERE <*Условие для выбора строки, содержащей объект*>;

— Например, получение ссылки на объект, находящийся в первой строке таблицы authors_2

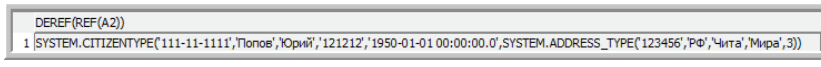
```
SELECT REF (a2) FROM authors_2 a2
where a2.au_id ='111-11-1111';
```



	REF(A2)
1	SYSTEM.CITIZENTYPE('111-11-1111','Попов','Юрий','121212','1950-01-01 00:00:00.0',SYSTEM.ADDRESS_TYPE('123456','РФ','Чита','Мира',3))

— Обратная по отношению к REF () функция DEREF (<Ссылка>) разрешает ссылку, возвращая сам объект:

```
SELECT DEREF (REF (a2))FROM authors_2 a2
    where a2.au_id ='111-11-1111';
```



	DEREF(REF(A2))
1	SYSTEM.CITIZENTYPE('111-11-1111','Попов','Юрий','121212','1950-01-01 00:00:00.0',SYSTEM.ADDRESS_TYPE('123456','РФ','Чита','Мира',3))

— Для получения свойства объекта по ссылке используется точечная запись DEREF (<Ссылка >).<Имя свойства>. Например:

```
SELECT DEREF (REF (a2)).au_lname FROM
authors_2 a2
where a2.au_id ='111-11-1111';
```

Работа со ссылками в блоке PL\SQL

```
DECLARE
AVTOR CitizenType; — объявление объекта типа
CitizenType
AVTOR_REF REF CitizenType; — объявление ссыл-
ки на объект
BEGIN
```

— Чтение ссылки на объект, находящийся в строке таблицы БД:

```
SELECT REF (a2) INTO AVTOR_REF FROM authors_2 a2
where a2.au_id ='111-11-1111';
```

— Так как для чтения объекта по ссылке необходимо обращение в БД, прямое присваивание объекта, например AVTOR:= DEREF (AVTOR_REF); не допустимо.

— Чтение объекта из базы по ссылке требует использования таблицы DUAL:

```
SELECT DEREF (AVTOR_REF) INTO AVTOR FROM DUAL;
```

— Вывод атрибутов (фамилии и имени) объекта, полученного по ссылке:

```
DBMS_OUTPUT. PUT_LINE (AVTOR.au_lname||' '||
AVTOR.au_fname);
END;
```

Использование ссылок для связывания объектных таблиц

Ссылка, размещенная в поле таблицы, поддерживает связь наборов данных с объектами и связь между объектами в БД. Например, требуется связать данные, находящиеся в строке таблицы `authors_1` с определенным объектом в таблице `authors_2` (рис. 4.4). Для этого в таблице `authors_1` создается дополнительный столбец, в который включаются ссылки на объекты в таблице `authors_2`.

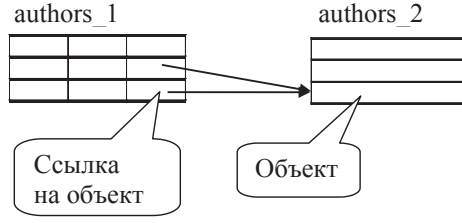


Рис. 4.4. Использование ссылок для связывания объектов

1. Добавление в `authors_1` столбца для хранения ссылок на объекты типа `CitizenType`:

```
Alter Table authors_1 add (AVTOR_Ref_Col Ref
CitizenType)
```

Запись ссылки на объект в поле таблицы выполняют обычные SQL-команды UPDATE или INSERT. При этом объект, на который создается ссылка, должен существовать в БД,

а ссылку на него можно получить подзапросом. Например, запись ссылки на объект, находящийся в строке *таблицы2* в ссылочном поле *таблицы1*:

```
UPDATE <Таблица1> SET <Ссылочное поле> =
(SELECT REF (<Алиас>) From <Таблица2> <Алиас>
WHERE <Условие выбора строки с объектом в таблице2>)
WHERE <Выбор строки для записи ссылки в таблице1>;
```

— Запись в столбец AVTOR_Ref_Col *таблицы authors_1 ссылок на все объекты из таблицы* authors_2 *(соотношение строк в таблицах 1:1):*

```
UPDATE authors_1 a1
Set a1.AVTOR_Ref_Col =
(Select REF (a2) From authors_2 a2
WHERE a1.AUTHOR.au_id = a2.au_id);
```

— Проверка обновления ссылочного столбца:

```
SELECT * FROM authors_1;
```

	ID_AUTH	NICK	AUTHOR	AVTOR_REF_COL
1	1	Первый	[SYSTEM.CITIZENTYPE]	SYSTEM.CITIZENTYPE('111-11-1111','Попов','Юрий','121212','1950-01-01 00:00:00.0',SYSTEM.ADDRESS_TYPE('123456','РФ','Чита','Мира',3))
2	2	Второй	[SYSTEM.CITIZENTYPE]	[SYSTEM.CITIZENTYPE]

2. Использование ссылок для доступа к объектам в БД в программе PL\SQL.

Программа получает объект из таблицы authors_2 по ссылке, извлекаемой запросом из таблицы authors_1.

```
DECLARE
AVTOR_REF REF CitizenType; — переменная для ссылки AVTOR CitizenType; — для чтения объекта из БД
BEGIN
```

— Чтение ссылки из поля AVTOR_Ref_Col таблицы authors_1 в AVTOR_REF:

```
SELECT a1.AVTOR_Ref_Col INTO AVTOR_REF FROM
authors_1 a1
```

```
   where a1.AUTHOR.au_id ='222-22-2222';
```

— Получение объекта по ссылке:

```
   SELECT DEREF (AVTOR_REF) INTO AVTOR FROM DUAL;
```

— Вывод атрибутов объекта, полученного по ссылке:

```
   DBMS_OUTPUT. PUT_LINE (AVTOR.au_lname || ' '||
AVTOR.au_fname);
   END;
```

Рассмотренные средства позволяют сделать вывод о том, что объектные типы Oracle во многом соответствуют возможностям объектно-ориентированных баз данных.

5. Документная база данных MongoDB

Независимо от модели данных большинство традиционных СУБД требуют создания описания хранящихся в базе данных. Такие описания называют схемами или словарями данных. В схемах представлены структуры и ограничения данных, схемы используются при выполнении запросов и обновляющих команд. Динамическая природа хранимых объектов вызывает изменение состава существующих характеристик, что требует внесения изменения в схемы прежде, чем данные могут быть включены в БД. Поэтому жесткие схемы для баз данных становятся препятствием в развивающихся информационных системах, которым необходимо быстро адаптироваться к изменению состава данных. Отсутствие схемы базы данных обеспечивает легкость ее создания и гибкость в использовании. Однако при отсутствии схемы СУБД уже не может обеспечить некоторые виды контроля данных и разделение между хранимыми структурами данных и их абстракциями уровня приложений. Кроме того, в отсутствие схемы для сохранения возможности доступа к данным по их именам необходимо каждое данное сопровождать его названием, что приводит к избыточности при вводе данных и требует знания этих имен пользователями БД. Необходимость именования каждого атрибута требует использования в составе информационного объекта конструкций <*Имя данного* ><*Значение данного*>, которые при значительной вложенности и повторяемости данных делают информацию об объекте трудночитаемой.

В попытках создать бессхемную СУБД наибольшего успеха добились разработчики компании 10gen в продукте MongoDB. Сервер БД MongoDB написан на языке C++ и выпускается с открытым исходным кодом. Использование развитой документо-ориентированной модели с большим числом возможностей для представления данных и свободное распространение MongoDB в рамках лицензии Creative Commons обусловили его широкое внедрение в информационные системы. В настоящее время СУБД MongoDB реализована во многих операционных системах, включая облачную Windows Azure. Для работы с базой MongoDB созданы драйверы для большинства языков программирования: PHP, Ruby, Python, C#, C++, Java [12].

5.1. Модель данных в MongoDB

Основной единицей хранения, доступа и обработки в базе MongoDB является документ. Документ имеет набор свойств — атрибутов. Каждый атрибут в документе задается парой *<Имя атрибута> : <Значение атрибута>*.

Наборы документов хранятся в коллекциях. Хотя в одну коллекцию обычно помещаются документы, представляющие однотипные объекты, документы в одной коллекции не обязаны иметь одинаковые наборы атрибутов. Также не требуется совпадения типов значений одноименных атрибутов в разных документах коллекции.

Для идентификации каждый документ имеет атрибут с именем _id, содержащий уникальный ключ и набор заданных пользователем произвольных атрибутов. Если при включении документа ключ не задан, то он генерируется автоматически. Автоматически созданный ключ соответствует 24-разрядному шестнадцатеричному числу. В целом для задания документа используется формат JSON (JavaScript Object Notation). Спец-

ификация RFC 4627 на JSON применительно к MongoDB требует, чтобы каждый документ имел формат, представленный на рис. 5.1.

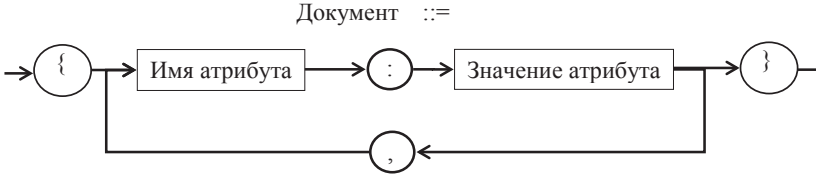


Рис. 5.1. Формат JSON-документа в базе MongoDB

Документ — неупорядоченный набор пар <*Имя (ключ) атрибута*> : <*Значение атрибута*>. Документ начинается с открывающей фигурной скобки и заканчивается закрывающей фигурной скобкой. Имя и значение атрибута разделяет двоеточие. Пары <*Имя атрибута*> : <*Значение атрибута*> в документе разделяются запятой.

Значение атрибута может иметь простой предопределенный в системе тип. Простые (атомарные) типы данных представлены в табл. 5.1.

Таблица 5.1

Тип	Значение	Пример записи
string	"*Строка*" или '*Строка*', где Строка — последовательность из нуля или больше символов Unicode. Символы задаются односимвольными строками	"Первый", '0'
numbers	Десятичные числа	100, 3.14
Boolean	true false	
Date	Дата или дата и время по Гринвичу	ISODate ("1831-02-16") ISODate ("2015-01-31T10:33:10.550Z")

Окончание табл. 5.1

Тип	Значение	Пример записи
ObjectId	24-байтовое, шестнадцатеричное целое число	Используется для идентификации документов _id: ObjectId ("559a8c8d250c6e372a67abbd")
Признак отсутствия значения	`null`	

Кроме простого типа значение атрибута может быть представлено массивом значений или другим документом. Таким образом, атрибуты в документах могут образовывать вложенные структуры. Массив — упорядоченный набор значений (рис. 5.2). Массив начинается с открывающей квадратной скобки и заканчивается закрывающей квадратной скобкой. Элементы массива разделены запятой и индексируются целыми числами начиная с нуля. Вложенные массивы не допускаются. Однако если элемент массива является документом, то его атрибуты также могут быть заданы массивами.

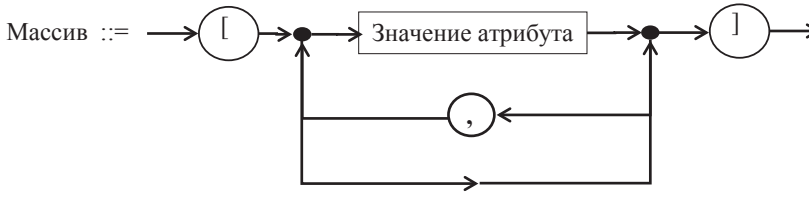


Рис. 5.2. Формат задания массива

В документе между любыми лексемами можно задавать произвольное число пробелов. Пример документа, содержащего сведения об авторе Джеймсе Олдридже и заданных в массиве Books двух его книгах:

```
  {Au_id: "000-11-0001", Au_lname: "Олдридж,
Джеймс", Years: [1918, 2015], Contract: false,
```

```
Books: [{Title_id: "BB1111", Title: "Морской
орел"},
{Title_id: "BB2222", Title: "Последний дюйм"}]}.
```

При добавлении документа в базу указывается коллекция, в которую включается и где хранится документ. Коллекции в БД должны иметь уникальные имена. Управление документами выполняется функциями (методами) коллекции. Для поиска, включения, удаления или изменения документа необходимо вызвать соответствующий метод коллекции. Коллекции являются объектами БД и имеют полный набор методов для доступа и обработки документов.

В целом база данных в MongoDB — набор созданных пользователями и служебных коллекций. Структура основных понятий MongoDB и их примерное соответствие терминологии реляционных баз представлены на рис. 5.3.

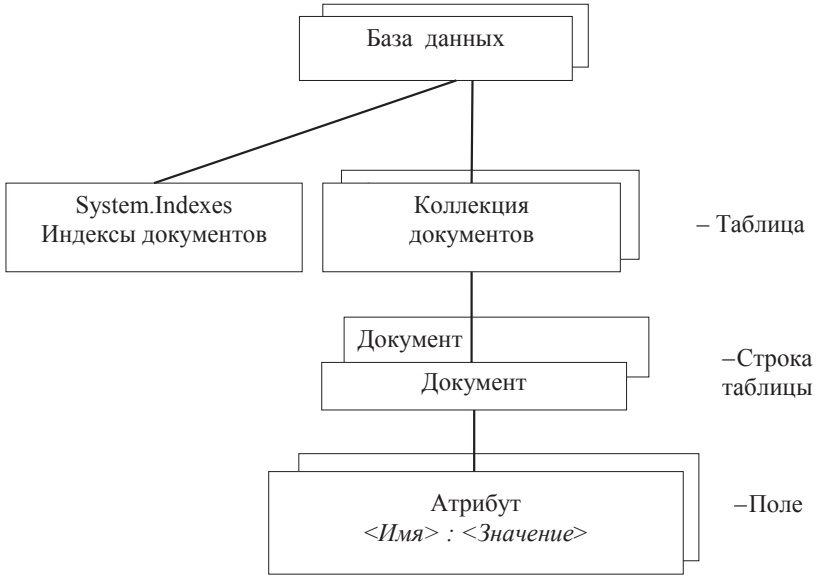


Рис. 5.3. Структура основных понятий базы в MongoDB

Кроме информационных в документе могут быть заданы специальные атрибуты-ссылки на другие документы в этой же или другой коллекции. С помощью ссылок в базе MongoDB поддерживается связь между документами. Возможности связывания документов, находящихся в одной и разных коллекциях посредством создания атрибутов-ссылок, показаны на рис. 5.4.

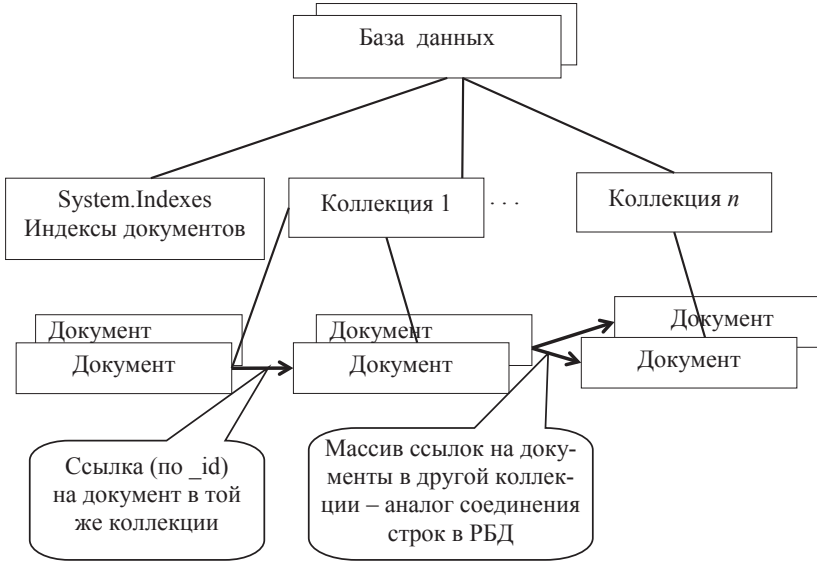


Рис. 5.4. Связывание документов посредством ссылок

5.2. Конфигурирование и запуск MongoDB в среде Windows

В процессе установки MongoDB в домашнем каталоге (папка, выбранная для установки) размещаются файл readme, файл описания условий лицензии (GNU-AGPL-3.0) и создается папка bin, содержащая исполняемые файлы: mongod.exe — сама

СУБД, mongo.exe — консольное приложение для работы с БД и дополнительные утилиты для работы администратора.

Для файлов данных и журналов необходимо иметь отдельные каталоги. Например, в домашнем каталоге MongoDB (`D:\mongodb\`) создаются:

- `databases` — папка для размещения баз данных;
- `log` — папка для журналов запуска экземпляра MongoDB.

Сведения о месте нахождения баз данных и журнала должны быть заданы в файле конфигурации `mongod.cfg`. Текстовый файл `mongod.cfg` находится домашнем каталоге MongoDB. В файле `mongod.cfg` в параметрах `dbpath` и `logpath` указываются пути к каталогам для размещения баз данных и файлов журнала. Например:

```
#Каталог для размещения баз данных
dbpath= D:\mongodb\databases
#каталог для файлов журнала
logpath= D:\mongodb\log\mongo.log
#устанавливает фиксацию ошибок данных в журнале
diaglog=3
```

Для других параметров при работе с локальным сервером могут быть использованы значения по умолчанию.

Локальная работа с БД MongoDB возможна путем запуска процесса `mongod` из каталога `bin` с параметром — `config`, задающим конфигурационный файл. Например:

```
D:\mongodb\mongod.exe — config D:\mongodb\mongod.cfg.
```

Однако для mongod.exe более удобно создать автоматически запускаемую службу Windows. Для этого необходимо в командном окне (`cmd`) с правами администратора:

- установить службу MongoDB с указанием используемого конфигурационного файла, например:
  ```
  D:\mongodb\mongod.exe — config D:\mongodb\mongod.cfg
  ```

— install;
- стартовать службу mongod командой net start MongoDB.

По умолчанию MongoDB будет прослушивать подключения клиентов по порту 27017.

Начиная с версии 2.6 изменен формат конфигурационного файла. Теперь опции сервера в файле mongod.conf записываются в формате YAML. Так, задание путей к каталогам для файлов журнала и баз данных в версии MongoDB 3.0.4 имеет вид:

```
## каталог для журналов
systemLog:
destination: file
path: "D:/_upp/MongoDB304/Server/3.0/log/mongo.log"
logAppend: true
## каталог для баз данных
storage:
dbPath: "D:/_upp/mongoDB304/Server/3.0/data-
bases"
directoryPerDB: true
journal:
enabled: true
```

5.3. Средства для работы с базой данных под управлением MongoDB

Для работы с базой в MongoDB поставляется консольное приложение Mongo.exe для ввода командных строк Mongo Shell. В строке Mongo Shell можно вводить любые команды и вызывать методы коллекций БД. Результаты их выполнения сохраняются в базе или выводятся в окне Mongo Shell. Mongo Shell является основным инструментом создания и управления базами MongoDB.

Типовые задачи администрирования БД выполняют утилиты MongoDB:

- `mongoexport.exe` и `mongoimport.exe` — утилиты вывода и загрузки документов из/в коллекции в формате JSON;
- `mongodump` и `mongorestore` — утилиты копирования и восстановления базы.

На сайте MongoDB по адресу http://docs.mongodb.org/ecosystem/drivers/можно найти драйверы для работы с БД из наиболее популярных языков программирования. Различные средства работы с базой MongoDB показаны на рис. 5.5.

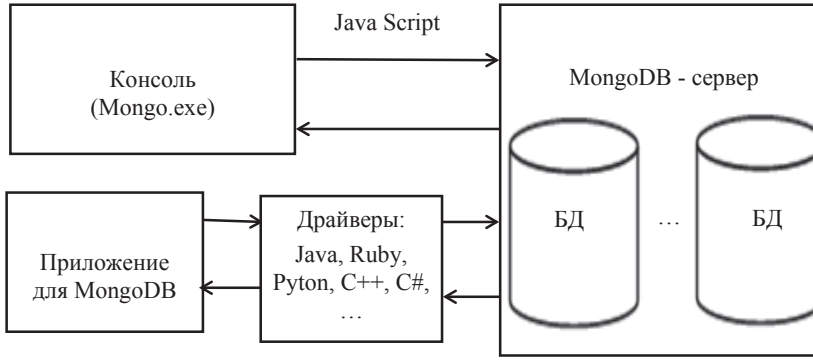


Рис. 5.5. Средства для работы с базой данных MongoDB

Кроме штатных средств управления данными и разработок для баз MongoDB, имеются продукты сторонних фирм. Например, разнообразный набор интерактивных средств для работы с MongoDB предоставляет NoSQL Manager for MongoDB Professional.

5.4. Работа с базой в консоли Mongo

Для работы с БД в `Mongo Shell` необходимо запустить файл `mongo.exe`, находящийся в папке *<Домашний каталог>*`\bin`, например `D:\mongodb\mongo.exe`. Происходит подключение к серверу MongoDB и устанавливается умалчиваемая база `test`, к которой адресуются последующие запросы. Для работы с БД может быть запущено несколько таких приложений. Далее в ответ на приглашение «>» в командной строке вводятся команды и обращения к методам управления данными, а также дополнительные команды для получения справок и выполнения функций операционной системы.

Справочная команда `help` выводит сведения о назначении других команд и обращения к справочникам по методам отдельных объектов MongoDB. Например, информацию о методах базы данных, имеющей псевдоним `db`, можно получить обращением к методу `db.help ()`, а выполнение метода `db.`*<Имя коллекции>*`.help ()` предоставит информацию о методах коллекции.

Команды, необходимые для начала работы с MongoDB, представлены в табл. 5.2.

Таблица 5.2

Команда	Назначение
Команды и методы для работы с БД в MongoDB Shell	
`help`	Вывод списка команд MongoDB Shell и справочных методов для БД
`db.help ()`	Справочник по методам базы данных
`db.`*<Имя коллекции>*`help ()`	Справочник по методам коллекции
`show dbs`	Вывод имен, имеющихся на сервере БД

Окончание табл. 5.2

Команда	Назначение
use <*Имя БД*>	Установка текущей БД, с которой далее ведется работа. Установить можно и несуществующую базу. Создание первой коллекции приведет к созданию базы. Обращение к установленной базе выполняется под псевдонимом >db. Например, >db.getCollectionNames () выполняет метод БД, возвращающий имена коллекций в установленной базе
db.getName ()	Вывод имени текущей базы
show collections	Вывод имен коллекций в установленной базе (команда приложения mongo shell)
show logs	Вывод имен используемых журналов. По умолчанию имя журнала global
show log [<*Имя журнала*>]	Возвращает из журнала сведения о параметрах подключения к серверу, конфигурационном файле, имеющихся БД. По умолчанию имя журнала global
db.dropDatabase ()	Удаление базы
exit	Выход из mongo shell
Команды создания и обработки коллекции в БД	
db.createCollection (<*Имя коллекции*>. [,{size:… [, capped: …, max: …}]])	Создание новой коллекции
db.getCollectionNames ()	Вывод списка имен коллекций в виде массива.
db.<*Имя коллекции*>.find ()	Вывод документов из коллекции
db.<*Имя существ. коллекции*>. renameCollection ("<*Новое имя коллекции*>")	Переименование коллекции
db.<*Имя коллекции*>.drop ()	Удаление коллекции
db.<*Имя коллекции*>.find ()	Вывод всех документов из заданной коллекции

Создание и удаление коллекций

Создание новой коллекции выполняет метод `createCollection` для БД

```
>db.createCollection (<Имя коллекции>
  [, {size:<Размер базы в байтах>
  [, capped: true|false, max:<Число документов>}]])
```

Например, создание коллекции `authors`

`>db.createCollection ("authors");` приводит к созданию определения коллекции `authors`. Фактическое создание коллекции происходит при включении в нее первого документа.

Выполнение метода

`>db.createCollection (authors, {size: 3000000})` создает коллекцию `authors`, для которой предварительно резервируется ~3 Мб в каталоге для баз данных, заданным параметром `dbpath` в конфигурационном файле `mongod.cfg`.

Количественные характеристики коллекции выводит ее метод `stats ()`:

>db.<*Имя коллекции*>.stats ();

Например, метод `db.authors.stats ();` выводит статистику коллекции `authors` в базе `Authors` (размеры букв в именах имеют значения):

```
"ns": "Authors. authors",
"count": 0,
"size": 0,
"storageSize": 3002368,
"numExtents": 1,
"nindexes": 1,
"lastExtentSize": 3002368,
"paddingFactor": 1,
"systemFlags": 1,
"userFlags": 0,
"totalIndexSize": 8176,
```

```
"indexSizes": {
    "_id_": 8176
},
"ok": 1
```

В строке `"storageSize": 3002368` указан размер зарезервированной памяти.

Необязательный параметр `capped: true` метода `db.createCollection` используется при создании ограниченных коллекций, в которых общее число документов не может превышать значение, заданное параметром `max:<`*`Число документов`*`>`. При добавлении в ограниченную коллекцию лишних документов происходит удаление наиболее старых документов. Ограниченные коллекции требуют обязательного указания размера резервируемой памяти `(size)` и не допускают добавления в существующие документы новых атрибутов.

Вывод списка имеющихся в базе коллекций выполняет метод `db.getCollectionNames ()`.

Для полного удаления коллекции со всеми имеющимися в ней документами предназначен метод самой коллекции `drop`. Например, `>db.authors.drop ()` удалит коллекцию `authors` из текущей БД.

Переименование коллекции выполняется методом `renameCollection: db.<`*`Имя существующей коллекции`*`>.renameCollection ("<`*`Новое имя коллекции`*`>")`.

Пример переименования коллекции `OldCollection`:

```
db.OldCollection.renameCollection
("NewCollection");
```

Включение документов в коллекцию

Добавление нового документа в коллекцию выполняет метод `Insert ()`:

`db.<`*`Имя коллекции`*`>.insert (<`*`Документ в формате JSON`*`>)`

или db.<*Имя коллекции*>.insert ({<*Имя атрибута*>:<*Значение*> [,..});

Например, включение в коллекцию авторов Николая Лескова может выполнить метод:

```
db.authors.insert ({"Au_lname":"Лесков", "Au_fname": "Николай", "birthday": ISODate ("1831-02-16")});
```

При этом атрибут "birthday" (день рождения) имеет тип Date и поэтому задается строкой в аргументе функции ISODate, которая преобразует строку в значение типа дата-время в формате GMT. В примере время в течение суток не задано, поэтому будет доопределено нулем (началом суток).

Запрос для вывода документа по условию (селектору) на фамилию автора db.authors.find ({"Au_lname": "Лесков"}) возвратит весь документ:

```
{"_id": ObjectId ("559a8c8d250c6e372a67abbd"),
"Au_lname": "Лесков", " Au_fname ": "Николай",
"birthday": ISODate ("1831–02–16T00:00:00Z")}
```

В момент включения документа сервер MongoDB автоматически создал для него ключевой атрибут _id, имеющий специальный тип ObjectId с уникальным значением ObjectId ("559a8c8d250c6e372a67abbd").

Ключ документа может использоваться в условии запроса для получения конкретного документа. Например, запрос db.authors.find ({_id: ObjectId ("559a8c8d250c6e372a67abbd")}) возвратит тот же самый документ.

Пользователь при добавлении документа методом insert может задать свое уникальное в коллекции значение атрибута _id. Например,

```
db.authors.insert ({_id: "000-11-1111", "Au_lname": "Пелевин", " Au_fname ": "Ви́ктор",
"year_of_birth": 1962, Contract: true});
```

По заданному пользователем значению _id также возможен поиск документа: db.authors.find ({_id: "000-11-1111"});

Метод insert позволяет добавлять в коллекцию документы из переменной java script:

```
var doc={"Au_lname":"Н.В. Гоголь", "year_of_birth":1809};
db.authors.insert (doc);
```

Другой способ наполнения коллекции предоставляет утилита mongoimport.exe. Источником данных для mongoimport служит текстовый файл. Кодировка символов в файле-источнике должна быть в формате UTF-8, преобразующем 16-битные символы Юникода в 8-битные. Имя загружаемой базы, коллекции и исходный файл задаются значениями ключей в вызове утилиты. Например, пусть файл ImportFile.json содержит два документа для загрузки в коллекцию authors базы Authors:

```
{"_id":"01", "Фамилия": "Калинин", "Имя":"Юрий",
"Книги": ["Букварь","Чтение"]}
{"_id":"02", "Автор": "Сумкин Н.", "Книги":
["Азбука"]}.
```

Загрузку документов из файла выполнит утилита:

```
<Путь>\mongoimport -d Authors -c authors
    <Путь>\ImportFile.json
```

Несмотря на различие в составе и названиях атрибутов, новые документы будут добавлены в существующую коллекцию authors.

Документы в загружаемом файле могут быть заданы не только в JSON, но и в форматах, предназначенных для представления таблиц: csv, tsv. Справку по всем ключам утилиты можно получить ее вызовом с ключом — help: mongoimport.exe — *help*.

Для вывода документов из коллекции в текстовый файл предназначена утилита mongoexport.exe.

Средства поиска данных в базе MongoDB

Целью запроса к базе MongoDB являются документы из определенной коллекции. Для поиска (отбора) документов предназначены методы коллекции find и findOne. Вызовы методов find и findOne имеют одинаковую структуру и содержат критерий отбора документов (селектор), имена выводимых атрибутов и правило сортировки выбранных документов. Различие методов в том, что метод find возвращает все документы, соответствующие критерию отбора, а метод findOne — только один (первый) документ из удовлетворяющих селектору.

Основной синтаксис вызова метода find () имеет вид:

```
db.<Коллекция>.find ([{<Селектор>}
[, {<Список атрибутов>}]]);
```

В простейшем виде запрос db.<*Коллекция*>.find (); возвращает все документы заданной коллекции. Отсутствие селектора и списка атрибутов приводит к выводу всех документов с полными наборами их атрибутов. Параметр <*список атрибутов*> содержит имена выводимых или, напротив, исключаемых из вывода атрибутов найденных документов.

Кроме параметров find и findOne имеют присоединенные методы, выполняющие сортировку, ограничение количества и способ вывода найденных документов. С присоединенными методами запрос имеет вид:

```
db.<Коллекция>.find ([{<Селектор>}
[, {<Список атрибутов>}]])
[.sort ({<Атрибут>: 1 |-1, …})]
[.limit (<Число выводимых документов>)]
[.skip (<Количество пропускаемых документов>)]
[.pretty ()]; — выводит каждый атрибут в отдельной
```

строке.

Количество документов, удовлетворяющих запросу, возвращает метод коллекции count: db.<*Коллекция*>.**count** ({<*Селектор*>}).

Вывод различных значений (удаление дубликатов) определенного атрибута в коллекции выполняет метод db.<*Коллекция*>.**distinct** (<*Имя атрибута*>). Например, запрос db.authors.distinct ("Au_lname") вернет массив из фамилий авторов, выбранных из атрибута Au_lname в документах коллекции authors.

5.5. Селекторы в MongoDB

Селектор содержит критерий для выбора документов и записывается в формат JSON-объекта: {<*Условия на значения атрибутов*>}. Пустой селектор, который обеспечивает выборку всех документов из коллекции, задается отсутствием условия **{}** или *Null*. Отдельные условия и логические операции в селекторе также записываются в формате JSON.

Виды условий в селекторе

1. <*Атрибут*>:<*Значение*> соответствует условию <*Атрибут*>=<*Значение*>.

Например, поиск документа по фамилии автора:

```
db.authors.find ({"Au_lname": "Пелевин"});
```

2. Модификаторы сравнений в условии **$lt** (less than, **<**), **$lte** (less than or equal, **<=**), **$gt** (greater than, **>**), **$gte** (greater than or equal, **>=**), **$ne** (not equal, **#**) задают условие выбора документов в форме неравенства. Для сохранения формата условия модификатор и значение для сравнения записывается в виде JSON-объекта:

{<*Модификатор*>: <*Значение для сравнения*>}.

Например, db.authors.find ({"Au_lname": "Пелевин", "Au_fname": {$ne: "Виктор "}}.

Сравнению **$ne** (не равно) удовлетворяют и те документы, в которых проверяемый атрибут отсутствует. Перечисле-

ние нескольких условий требует их одновременного выполнения в выбираемых документах.

3. Инвертирование условия **$not.**

<*Атрибут*>:{$not: {<*Сравнение*>}}}. Модификатор **$not** применим только с модификаторами сравнения **$lt, $lte** и т. д. Условию с модификатором **$not** также удовлетворяют документы, не имеющие проверяемого атрибута. Например, поиск авторов, год рождения которых меньше или равен 1990:

```
db.authors.find ({"Год_рожд.": {$not: {$gte:1990}}});
```

4. **$regex** — регулярные выражения в селекторе (поиск по строковым атрибутам).

Краткая запись условия с регулярным выражением:

<*Строковый атрибут*>:/<*Шаблон*>/<*Опции* **i** *и* **m**>,

где шаблон — искомый контекст.

Например, запрос `db.authors.find ({Фамилия:/Кот/});` находит `"Котов"`, `"РоКотов"`, но не найдет `"Рокотов"`.

Опции: **i** — сравнение выполняется без учета регистра символов, **m** — сравнение применяется к многострочным текстам.

Например, `db.authors.find ({Фамилия:/ве/i});` находит: С**ве**тин, С**ве**тов, **Ве**тров.

Полная запись условия в регулярном выражении требует модификатора **$regex**: <*Атрибут*>: {$regex:/<*Шаблон*>/, $options: '<*Список опций*>'}

Для расширения условия поиска в шаблоне используются специальные символы:

- `?` — указанный в шаблоне знак вопроса обозначает, что этот элемент может как присутствовать, так и отсутствовать. Например, шаблон/ко?р/найдет 'кор' в слове "короткий" и 'кр' в — "краткий";
- выражение `x|y` находит x или y. Например, условие `{$regex:/Кор|кра/, $options:'i'}` также найдет слова "короткий" и "краткий";

- `[abc]` — набор символов. Находит любой из перечисленных символов. Набор символов можно задать промежутком в алфавите, используя тире. Например, [ийклм] — то же самое, что [и–м]. Условие `{$regex:/e [и-м]/, $options:'i'}` найдет слова "Пелéвин" и "Олдридж, Джеймс", содержащие перечисленные буквы за символом "е";
- запись `[^abc]` определяет любой символ, кроме указанных в наборе. Набор исключаемых символов можно также определить промежутком;
- символ "\" превращает обычный символ в специальный. Например, выражение/s/ищет просто символ "s", а\s ищет пробел. Выражение `{$regex:/\s/}` отбирает тексты, содержащие пробел. Наоборот, если символ специальный, например?, то\превращает его в обычный символ знак вопроса. Например,/\?/ищет тексты со знаком вопроса.

В MongoDB используются регулярные выражения JavaScript. Полный набор специальных символов JavaScript включает `*, {,}, +, $` и др. Их описание можно найти в [13].

5. **`$in`** — проверка совпадения значения атрибута с одним из элементов в массиве:

`{<`*`Атрибут`*`>: {$in: [<`*`Список значений`*`>]}.`

Например: `db.authors.find ({"Au_lname": {$in: ["Пелевин", "Лесков"]}}).`

6. **`$nin`** — проверка отсутствия определенных значений атрибута или самого атрибута (альтернатива для **`$in`**).

`<`*`Атрибут`*`>: {$nin: [<`*`Список значений`*`>]}` находит документы, в которых заданный атрибут не совпадает ни с одним из элементов в списке значений или документ не имеет проверяемого атрибута. Например, `db.authors.find ({"Au_lname": {$nin: ["Пушкин", "Лесков"]}})`

7. Выбор документов, одновременно удовлетворяющих нескольким условиям. В краткой форме отдельные условия задаются в общем списке:

<Атрибут1>:<Значение1>, <Атрибут2> : <Значение2>,...

Заданные списком условия связываются по «И». Например, запрос `db.authors.find ({"Au_lname": "Пелевин", "Au_fname": "Виктор"})` находит Виктора Пелевина.

Полная (в JSON) форма связывания по «И» использует модификатор `$and` с массивом условий: **`$and: [`***`{<Условие1>}, ...`***`]`**

Например, `db.authors.find ({$and: [{"Au_lname": "Пелевин"}, {"Au_fname": "Виктор"}]});`

8. Связывание условий по «ИЛИ» **`$or:`** `[`*`<Список условий>`*`]`. Например, `db.authors.find ({`**`$or:`** `[{"Au_lname":"Лесков"}, {"Au_lname":"Пелевин"}]});` находит автора Лескова или Пелевина.

9. Связывание условий по «инвертированному ИЛИ»: **`$nor: [`***`<Список условий >`***`]`** — выбирает документы, в которых не выполняются все условия или отсутствует атрибут, участвующий в условии. Например, запрос: `db.authors.find ({$nor: [{"Au_lname":"Лесков"},`

```
{"Au_fname":"Николай"}]});
```

выберет других авторов, кроме Николая Лескова, и авторов, не имеющих атрибута `Au_lname` или `Au_fname`.

10. Поиск документов по наличию/отсутствию атрибута:

<Атрибут>: **`{$exists: false|true}`**. По условию *<Атрибут>*: `{$exists: false}` находятся те документы, в которых проверяемый атрибут отсутствует или его значение не задано (`null`). Например, запрос `db.authors.find ({"Год_рожд":{$exists: false}})` находит авторов, у которых не задан атрибут "Год_рожд".

11. Проверка типа для значения атрибута:

`{<атрибут>: {$type: <`*`код BSON-типа`*`>}}`. Выбирает документы, у которых указанный атрибут имеет заданный тип. Основные коды BSON-типов: Double — 1 (числовой, задает целые числа и числа с фиксированной точкой), String — 2,

Boolean — 8, Date — 9, null — 10. Полный список кодов типов можно найти в [13]. Например, `db.authors.find ({"Год_рожд": {$type: 2}})` выберет документы, у которых атрибут "Год_рожд" задан строкой текста, а запрос `db.authors.find ({"Год_рожд": {$type: 10}})` выберет документы, у которых атрибут "Год_рожд" присутствует, но не задан (`null`).

Управление выводом атрибутов в запросе

Список выводимых атрибутов задается вторым параметром в методах `find` и `findOne`. В параметре атрибут, который необходимо вывести, указывается со значением 1, а не требуемый для вывода — 0.

`db.<`*коллекция*`>.find ([{<`*селектор*`>}`
`[, {<`*имя атрибута*`>: 1 |0, …}]);`

Например, `db.authors.find ({},{"Au_lname": 1, "Au_fname":1})` выведет из всех документов атрибуты `Au_lname` и `Au_fname` и ключ `_id`, который выводится по умолчанию, если его вывод не отключен явно.

Если в списке перечислены только атрибуты, не требующие вывода, то незаданные атрибуты выводятся по умолчанию. Например, `db.authors.find ({}, {_id: 0, "Год_рожд": 0})` выведет все атрибуты, кроме `_id` и `Год_рожд`, из всех документов.

В одном запросе не допускается смешивать выводимые и невыводимые атрибуты документа. Так, запрос `db.authors.find ({}, {"Au_lname": 1, "Au_fname": 0})` является ошибочным. Исключение составляет атрибут `_id` — идентификатор документов.

Сортировка документов в запросе

Для сортировки результата запроса `find ()` используется присоединенный метод `db.<`*Коллекция*`>.find ().sort ({<`*Атрибут*`>: 1 |-1,..})`. Указание атрибута со значением 1 приводит к сортировке по возрастанию, —1 — по убыванию значений атрибута.

Например, `db.authors.find ({}, {_id:0,"Au_lname": 1, "Au_fname": 1}).sort ({"Au_lname": 1})` выведет фамилии и имена авторов, отсортированные по возрастанию фамилии.

Ограничение множества выводимых документов

Для ограничения количества выводимых документов используются присоединенные методы:

- **limit** (<*Количество выводимых документов*>),
- **skip (<***Количество пропускаемых в начале документов***>)**.

Например, запрос `db.authors.find (null, {_id:0, "Au_lname":1}).limit (5).skip (2).sort ({"Au_lname": -1});` выведет из коллекции `authors` отсортированные по убыванию фамилии пяти авторов, пропустив две первых фамилии.

5.6. Удаление документов

Для удаления отдельных документов из коллекции предназначен перегружаемый метод `remove`. При вызове метода с одним параметром `db.<Коллекция >.remove ({<Селектор>})` происходит удаление всех документов, удовлетворяющих селектору. А в отсутствие селектора из коллекции удаляются все документы. При вызове с двумя параметрами `db.<Коллекция>.remove ({<Селектор>}, true)` удаляется только один (первый) документ из соответствующих селектору. Например, команда

```
db.authors.remove ({"Год_рожд":{$gt:2007}})
```

удалит документы с годом рождения авторов, большим, чем 2007.

5.7. Изменение документов

Изменения в документы коллекции вносятся методом `update`.

Метод может изменить определенные атрибуты одного или нескольких существующих документов или полностью заменить существующий документ новым. Вид выполняемого действия определяется используемыми параметрами и их модификаторами. Базовый вызов метода `update` имеет вид:

```
db. <Коллекция>.update ({<Селектор>},
{<Новый документ или модификатор с изменяемыми
атрибутами>}, {<Опции обновления>});
```

Полная замена документа, удовлетворяющего селектору, выполняется, если новые значения атрибутов заданы без использования модификаторов. Например: пусть в коллекции `authors` имеется документ:

```
{Au_id: "0007", lname: "Гоголь", Years: [1809,
1852]}
```

Применение метода

```
db.authors.update ({Au_id:"0007"}, {Au_
lname:"Гоголь", Years: [1809, 1852], Books:
["Вий", "Тарас Бульба"]})
```

приведет к замене выбранного селектором `{Au_id:"0007"}` документа новым.

Вместо нового документа вторым параметром метода `update` может быть задана переменная, содержащая документ.

Пусть имеется исходный документ: `{Au_id:"0007", lname:"Гоголь"}`

Заносим в переменную `doc` новый документ:

```
var doc = {Au_id: "0007", Au_lname: "Н.В. Го-
голь", Years: [1809, 1852]}
```

Выполняем замену документа:

```
db.authors.update ({Au_id: "0007"}, doc)
```

Третий параметр «Опции обновления» влияет на «поведение» метода `update`.

Опция `upsert (типа boolean)` со значением `true` при отсутствии документа, удовлетворяющие условию селектора, добавляет документ с заданными атрибутами. При значении `false` новый документ не создается. По умолчанию действует `false`.

Опция `multi (типа boolean)`: по значению `true` обновляются все документы, удовлетворяющего условию селектора. В значении `false` обновляет один документ. Значение по умолчанию `false`.

Добавление, удаление или изменение атрибутов в документе задается модификаторами, устанавливаемыми перед их значениями. Изменяемые атрибуты записываются в виде:

<Модификатор>:{*<Атрибут>*:*<Значение>*},..

Ниже приводится сводная таблица модификаторов, используемых в методе `update ()`.

Таблица 5.3

Модификатор	Описание
`$set`	Обновляет, а при отсутствии — создает атрибут
`$unset`	Удаляет атрибут
`$inc`	Увеличивает значение атрибута на заданное число
`$pop`	При значении 1 удаляет последний, при –1 — первый элемент массива
`$push`	Добавляет в массив новый элемент
`$pushAll`	Помещает несколько новых элементов в массив
`$addToSet`	Добавляет новый элемент в массив (исключаются дубликаты)
`$pull`	Удаляет из массива значение (при его наличии)
`$pullAll`	Удаляет из массива все подходящие значения

Рассмотрим использование модификаторов внесения изменений. Пусть в коллекции authors находится следующий документ:

```
{"Au_id": "000-11-0009", "LName": "Толстой",
"Lang": "Русский", "Years": [1882, 1945],
"Product": ["Аэлита", "Князь Серебряный"]}
```

1. Замена всего документа

```
db.authors.update ({Au_id: "000-11-0009"}, {Au_
id: "000-11-0009", Au_lname: "Толстой", Years:
[1882, 1945], Books: ["Аэлита", "Князь Серебря-
ный"]})
```

Запрос `db.authors.find ({Au_id: "000-11-0009"},{_id:0})` выведет новый документ:

```
{"Au_id": "000-11-0009", "Au_lname": "Толстой",
"Years": [1882, 1945],
"Books": ["Аэлита", "Князь Серебряный"]}
```

Дальнейшие изменения будут применены к новому документу.

2. Добавление нового атрибута. Модификатор `$set` добавляет атрибут, если его не было в документе:

```
db.authors.update ({Au_id: "000-11-0009"},
{$set:{"Страна": "СССР"}})
```

3. Изменение значения атрибута. Модификатор `$set` изменяет значение атрибута, если такой атрибут существует в документе. Уточняем автора:

```
db.authors.update ({Au_id: "000-11-0009"},
{$set:{Au_lname: "А.Н. Толстой"}})
```

После добавления нового и изменения существующего атрибута имеем:

```
{"Страна": "СССР", "Au_id": "000-11-0009",
"Au_lname": "А.Н. Толстой", "Books": ["Аэлита",
"Князь Серебряный"], "Years": [1882, 1945]}
```

4. Удаление ошибочного элемента "Князь Серебряный" из атрибута-массива "Books":

```
db.authors.update ({Au_id: "000-11-0009"},
{$pull:{"Books":"Князь Серебряный"}})
```

5. Добавление нового элемента "Петр Первый" в массив "Books"

```
db.authors.update ({Au_id: "000-11-0009"},
{$push:{"Books": "Петр Первый"}})
```

После обработки массива "Books" имеем исправленный документ:

```
{"Страна": "СССР", "Au_id": "000-11-0009",
"Au_lname": "А.Н. Толстой", "Books": ["Аэлита",
"Петр Первый"], "Years": [1882, 1945]}
```

5.8. Добавление или замена документа в коллекции — метод save

```
db.<Коллекция>.save (<Документ>)
```

Если указанный в параметре документ не содержит атрибут `_id`, то выполняется добавление документа с созданием уникального идентификатора `_id`.

Если атрибут `_id` в документе задан и он совпадает с `_id` документа в коллекции, то выполняется замена существующего документа, иначе добавляется новый документ с заданным ключом `_id`.

По аналогии с методами `insert` и `update` параметром метода `save` может быть содержащая документ переменная. Например,

```
var doc = {Au_id: "0008", Au_lname: "А.С. Пушкин", Years: [1799, 1837]};
db.authors.save (doc);
```

5.9. Использование переменных в скриптах обработки коллекций

В программах обработки БД для сохранения результатов предыдущих запросов, чтобы использовать их в последующих командах, используются переменные JavaScript.

Тип переменной определяется типом присвоенного значения.

[var] <*Имя переменной*> [= <*Константное выражение*>], ...

Присваивание значения переменной:

```
>var x = 90.99;
>doc={Имя:'Игорь', Фамилия:'Серый', Место_работы:'УрФУ'};
```

Чтение документа из коллекции в переменную выполняется методом `findOne`: `var <Имя переменной> = db.<Коллекция>.findOne (…);`

Для доступа к атрибуту документа, находящегося в переменной, используется точечная нотация: <*Имя переменной*>.<*Атрибут документа*>.

Например, изменение идентификатора документа в переменной:

```
<Имя переменной>._id =
new ObjectId ("51205738979cfae8e75bd6b0");
```

где "51205738979cfae8e75bd6b0" — шестнадцатеричный идентификатор документа (24 знака).

Добавление в коллекцию документа из переменной:

```
db.<Имя коллекции>.insert (<Имя переменной>);
```

Курсоры

Создание курсора из коллекции:

var <*Курсорная переменная*> = db.<*Коллекция*>.find (...);

Например, `var v = db.authors.find ({}, {_id:0});`

Доступ к документу в курсоре выполняется по индексу: 0, 1, 2, ...

Например, `v [1];` — обращение ко второму документу в курсоре, а обращение `v [1].Фамилия;` — доступ к атрибуту «Фамилия» второго документа.

Методы курсора для последовательной обработки документов

<*Курсорная переменная*>**.next ()** выполняет переход на следующий документ в курсоре и возвращает его. При первом выполнении метода возвращается первый документ.

<Курсорная переменная>`.hasNext ()` — проверка возможности перехода к следующему документу. Метод возвращает `true` при наличии и `false` при отсутствии следующего документа в курсоре.

Например, создание курсора:

```
var curs = db.authors.find ();
```

Последовательная обработка документов в курсоре:

```
while (curs.hasNext ()){
//Чтение следующего документа
d1 = curs.next ();
//Вывод атрибута «Фамилия» из прочитанного документа
print (d1.Фамилия);
//Здесь может быть фрагмент программы
//последовательной обработки документов в курсоре
.....
};
```

5.10. Группировка документов коллекции

Несложную группировку документов, аналогичную параметру group by в Select-SQL, выполняет метод `group`:

```
db.<Коллекция>.group (
{key: {<Ключ группировки>:1, …},
cond: {<Селектор>}},
reduce: function (Cursor, result) {<Агрегатная функция, выполняемая для каждого документа в группе>},
initial: {<Инициализация значений для группы>}
});
```

Параметр `key` содержит список атрибутов — ключей, по которым выполняется группировка документов, `cond` — условие выбора из коллекции документов для группировки, `reduce` со-

держит функцию, применяемую к каждому документу в группе. Параметры функции: `Cursor` — ссылка на обрабатываемый документ, `result` — ссылка на результаты — агрегатные значения, рассчитываемые в группе. Параметр `initial` устанавливает исходные значения для результата перед обработкой документов в группе.

Например, пусть в коллекции `Writer` находятся пять документов, содержащих данные о писателях и их книгах:

```
{"Автор": "Кант И.", "Страна": "Германия", "Название": ["Критика чистого разума"], "Тип": "Философия"}
{"Автор": "Пелевин В.О.", "Страна": "Россия", "Название": ["Чапаев и Пустота"], "Тип": "Художественная литература"}
{"Автор": "Пелевин В.О.", "Страна": "Россия", "Название": ["Бетман Аполло"], "Тип": "Фантастика"}
{"Автор": "Лукьяненко С.В.", "Страна": "Россия", "Название": ["Ночной дозор", "Дневной дозор"], "Тип": "Фантастика"}
{"Автор": "Пушкин А.С.", "Страна": "Россия", "Произведение": [["Руслан и Людмила", "Стихи"], ["Капитанская дочка", "Проза"]]}
```

Выполним группировку для подсчета числа авторов, для каждой страны и каждого типа произведения:

```
db.Writer.group ({
key: {"Страна": 1, "Тип": 1},
cond: {},
reduce: function (Cursor, result) {result.Число_авторов += 1},
initial: {Число_авторов: 0}})
```

В результате формируется массив документов. Каждый результирующий документ соответствует группе исходных документов с одинаковыми значениями ключевых атрибутов.

В результирующий документ вместе с ключевыми включаются атрибуты, рассчитанные функцией агрегатной обработки `reduce` для группы исходных документов.

В примере для групп документов с одинаковыми значениями атрибутов `"Страна"` и `"Тип"` вычисляется атрибут `"Число_авторов"`:

```
[{"Страна":"Германия","Тип":"Философия",
"Число_авторов":1},
{"Страна":"Россия", "Тип":"Художественная литература",
"Число_авторов": 1},
{"Страна":"Россия", "Тип":"Фантастика",
"Число_авторов":2},
{"Страна":"Россия", "Тип": null, "Число_авторов":1}]
```

5.11. Конвейерная обработка документов коллекции

В MongoDB реализован инструмент для выполнения цепочки операторов обработки документов коллекции — фреймворк агрегирования данных [14]. На вход агрегирования поступают исходные документы коллекции, для которых последовательно выполняется цепочка операций. Документы на выходе одной операции поступают на вход следующей. Результат последней операции является результатом всей обработки. Конвейерную обработку выполняет метод `aggregate`:

`db.<`*`Коллекция`*`>.aggregate ([<`*`Операция агрегации`*`>,..]);¶`

<Операция агрегации> содержит оператор — выполняемое действие и аргументы — атрибуты обрабатываемых входных документов, записанные в форме JSON-объекта:

{<*Оператор*>: [<*Аргумент1*>, < *Аргумент2*> …]}
или для одиночного аргумента операция агрегации имеет вид:
{<*Оператор*>: <*Аргумент*>}

Метод `aggregate` возвращает единственный документ с атрибутом `"result"`, содержащий массив документов, элементы которого являются результатом агрегатной обработки исходных документов.

В цепочке могут использоваться следующие операции агрегации.

1. {`$match`: {<*Селектор*>}}

Оператор `$match` фильтрует входные документы, отбирая соответствующие заданному селектору. В селекторе используются те же условия отбора, что и в методе `find`.

2. {`$project`: {<*Спецификация атрибута*>,...}}
управляет набором атрибутов в выходных документах, добавляя новые, удаляя и переименовывая атрибуты входных документов.

<*Спецификация атрибута*> имеет следующие формы:

- <*Атрибут входного документа*>:1 включает атрибут входного документа в выходной документ;
- _*id: 0* исключает вывод идентификатора документа (по умолчанию идентификаторы выводятся);
- <*Новый атрибут* >: <*Выражение для значения атрибута*> используется для создания новых атрибутов, значения которых вычисляются из атрибутов входного документа.

<*Выражение для значения атрибута*> записывается в форме JSON-объекта, содержащего массив:
{<*Операция*>: [<*Операнд1*>, <*Операнд2*>…]}
или {<*Операция*>: <*Единственный операнд*>}.

Тип операции зависит от типа операндов. Поддерживаются следующие типы операций:

- булевские (`$and, $or, $not`),
- строковые (`$concat, $substr, $toUpper` и т.д.),
- арифметические (`$add, $multiply` и т.д.),

- операции сравнения (`$eg, $gt, $gte и т.д.`) и другие типы.

Например, в агрегации `db.authors.aggregate ({$match:{"Au_lname":/Лес/}},`
`{$project:{_id: 0,`
`"Автор": {$concat: ["$Au_lname","  ","$Au_fname"]},`
`"birthday":1}});`
задан конвейер из двух операций:

- `$match` выбирает из коллекции `authors` документы об авторах, в фамилии которых присутствует слог «Лес», и передает следующей операции;
- `$project` для выбранных авторов отменяет вывод ключа `_id`, создает новый атрибут "Автор", в котором соединяются (`$concat`) в одной строке фамилия и имя автора, затем выводится атрибут "`birthday`". Операция `$concat` работает начиная с версии 3.0.4.

3. Операция {`$unwind:` *<Имя атрибута-массива>*} «разворачивает» массив, создавая для каждого элемента указанного массива новый документ, в котором присутствуют все атрибуты исходного документа с одним из элементов разворачиваемого массива.

4. Операция группировки входных документов

{`$group: {_id:` *<Группирующее выражение>*,
<Агрегирующий атрибут1>:
{*<Оператор агрегации1>*: *<Выражение агрегирования1>*},
`.....}}`

`$group` действует по аналогии с group by в SQL-Select: выполняет группировку путем создания нового документа из нескольких входных документов с одинаковым значением группирующего выражения и вычисления агрегирующих атрибутов по значениям атрибутов документов в группе. Способ вычисления агрегирующих атрибутов задается оператором агрегации.

Операторы агрегации:

- {$sum: <*Выражение*>} возвращает сумму числовых значений заданного выражения, вычисленного на документах в группе; нечисловые значения пропускаются;
- {$avg: <*Выражение*>} вычисляет арифметическое среднее числовых значений заданного выражения на документах в группе;
- {$first: <*Выражение*>} ({$last: < выражение >}) возвращает значение выражения, вычисленного по первому (последнему) документу в группе; используется, если определен порядок документов в группе;
- {$max: <*Выражение*>} ({$min: < выражение >}) возвращает максимальное (минимальное) значение выражения по документам в группе;
- {$push: <*Выражение*>} создает массив из результатов вычисления выражения для каждого документа в группе; порядок элементов в массиве не определен;
- {$addToSet: <*Выражение*>} создает массив из уникальных значений результатов вычисления выражения для каждого документа в группе.

5. Операция сортировки документов

{$sort: {<*Атрибут1*>: <*Порядок*>,, …}}

упорядочивает на выходе поток входных документов, выполняя ступенчатую сортировку по перечисленным атрибутам. Порядок сортировки задается значениями: 1 — сортировка по возрастанию атрибута, 0 — по убыванию.

6. Операция {$skip: <*Число документов*>} задает количество первых пропускаемых документов. Передает оставшиеся документы следующей операции в конвейере.

7. Операция {$limit: <Число документов>} ограничивает число документов, передаваемых следующей операции в конвейере.

8. Операция {$out: "<*Выходная коллекция*>"} записывает итоговые документы конвейера агрегации в заданную

коллекцию. Оператор `$out` должен быть последним этапом конвейера.

Рассмотрим пример конвейерной обработки коллекции документов. Пусть коллекция `Writer` содержит сведения о четырех писателях:

```
{Автор: "Кант И.", Страна: "Германия", Название:
["Критика чистого разума"], Тип:"Философия"},
{Автор: "Пелевин В.О.", Страна: "Россия", На-
звание: ["Чапаев и Пустота"], Тип:"Художественная
литература"},
{Автор: "Пелевин В.О.", Страна: "Россия", На-
звание: ["Бетман Аполло"], Тип:"Фантастика"},
{Автор: "Лукьяненко С.В.", Страна: "Россия",
Название: ["Ночной дозор", "Дневной дозор"],
Тип:"Фантастика"}.
```

Построим конвейер для подсчета числа книг каждого типа для русских писателей. Процесс обработки требует фильтрации документов по селектору {`Страна: "Россия"`} и последующей группировки по атрибуту `"Тип"`. В результате должны получиться документы следующего вида:

```
{"Тип": "Фантастика", "Количество": 3}.
```

Создание конвейера обработки по шагам

1. Выбор российских авторов.

```
db.Writer.aggregate ({$match:{Страна:/Рос/}});
```

Получим массив `"result"` из трех отфильтрованных документов:

```
{"result": [
{"_id": ObjectId ("55b8c9b5835c4398d7984cae"),
"Автор": "Пелевин В.О.", "Страна": "Россия",
"Название": ["Чапаев и Пустота"],
"Тип": "Художественная литература"},
{"_id": ObjectId ("55b8c9b5835c4398d7984caf"),
"Автор": "Пелевин В.О.","Страна": "Россия",
```

```
  "Название": ["Бетман Аполло"],
  "Тип": "Фантастика"},
  {"_id": ObjectId ("55b8c9bb835c4398d7984cb0"),
  "Автор": "Лукьяненко С.В.", "Страна": "Рос-
сия",
  "Название": ["Ночной дозор", "Дневной дозор"],
  "Тип": "Фантастика"}, "ok": 1}
```

2. В коллекции, полученной фильтрацией, разворачиваем массив названий книг.

```
db.Writer.aggregate ({$match:{Страна:/Рос/}},
{$unwind: "$Название"});
```

В результате для каждой книги формируется отдельный документ. Последний документ для автора Лукьяненко С. В., имеющего две книги, разворачивается в два документа этого автора:

```
{"result": [{"_id": ObjectId
("55b8c9b5835c4398d7984cae"),
  "Автор": "Пелевин В.О.", "Страна": "Россия",
  "Название": "Чапаев и Пустота",
  "Тип": "Художественная литература"},
  {"_id": ObjectId ("55b8c9b5835c4398d7984caf"),
  "Автор": "Пелевин В.О.", "Страна": "Россия",
  "Название": "Бетман Аполло", "Тип": "Фанта-
стика"},
  {"_id": ObjectId ("55b8c9bb835c4398d7984cb0"),
  "Автор": "Лукьяненко С.В.", "Страна": "Рос-
сия",
  "Название": "Ночной дозор", "Тип": "Фантастика"},
  {"_id": ObjectId ("55b8c9bb835c4398d7984cb0"),
  "Автор": "Лукьяненко С.В.", "Страна": "Россия",
  "Название": "Дневной дозор", "Тип": "Фантасти-
ка"}], "ok": 1}
```

Документы, содержащие несколько массивов, могут быть развернуты отдельно по каждому массиву или по нескольким любым массивам.

Например, пусть в коллекцию добавляется документ с двумя массивами, "Название" и "Формы":

```
db.Writer.insert ({Автор:"Пушкин А.С.", Страна: "Россия",
Название: ["Руслан и Людмила", "Капитанская дочка"],
Формы: ["Стихи", "Проза"]});
```

Тогда его последовательное разворачивание по обоим массивам

```
db.Writer.aggregate ({$match:{Страна:/Рос/}},
{$unwind: "$Название"}, {$unwind: "$Формы"});
```

приводит к созданию документов, в которых представлены все пары элементов из этих массивов:

```
{"result": [
{"_id": ObjectId ("55bc4ed307167e44cdcdd8a0"),
"Автор": "Пушкин А.С.", "Страна": "Россия",
"Название": "Руслан и Людмила",
"Формы": "Стихи"},
{"_id": ObjectId ("55bc4ed307167e44cdcdd8a0"),
"Автор": "Пушкин А.С.", "Страна": "Россия",
"Название": "Руслан и Людмила",
"Формы": "Проза"},
{"_id": ObjectId ("55bc4ed307167e44cdcdd8a0"),
"Автор": "Пушкин А.С.", "Страна": "Россия",
"Название": "Капитанская дочка",
"Формы": "Стихи"},
{"_id": ObjectId ("55bc4ed307167e44cdcdd8a0"),
"Автор": "Пушкин А.С.", "Страна": "Россия",
"Название": "Капитанская дочка",
"Формы": "Проза"}], "ok": 1}
```

Обратите внимание, в результат попадают только те документы, которые содержат разворачиваемые массивы. В приведенном примере разворачивание продемонстрировало формирование документов с семантическим несоответствием названия и формы произведения ("Название": "Капитанская дочка",

"Формы": "Стихи"). Причина несоответствия в отрыве связанных по смыслу данных при размещении названия и формы произведения в разных массивах. Для исключения подобных ошибок связанные данные должны размещаться в одном массиве. Например, можно представить сведения о названии и форме произведения в виде вложенного массива:

```
db.Writer.insert ({Автор:"Пушкин А.С.",
Страна:"Россия",
Произведение: [["Руслан и Людмила", "Стихи"],
["Капитанская дочка", "Проза"]]});
```

Теперь разворачивание внешнего массива "Произведение" дает верный результат:

```
db.Writer.aggregate ({$match:{Страна:/Рос/}},
{$unwind: "$Произведение"});
{"result": [
{"_id": ObjectId ("55bc55d707167e44cdcdd8a1"),
"Автор": "Пушкин А.С.", "Страна": "Россия",
"Произведение": ["Руслан и Людмила", "Стихи"]},
{"_id": ObjectId ("55bc55d707167e44cdcdd8a1"),
"Автор": "Пушкин А.С.", "Страна": "Россия",
"Произведение": ["Капитанская дочка", "Проза"]}], "ok": 1}
```

Так как новый документ о Пушкине А. С. не содержит массив "Название", он не повлияет на результаты следующих шагов.

3. Для подсчета количества книг каждого типа необходима группировка документов по типу книги. Для этого на третьем шаге в каждом документе, полученном разворачиванием массива "Название", оставляем необходимый для группировки атрибут "Тип" и добавляем новый атрибут "Количество" равным 1 во всех документах. На следующем шаге при группировке атрибут "Количество" будет суммироваться отдельно для каждого типа книг.

```
db.Writer.aggregate ({$match:{Страна:/Рос/}},
{$unwind: "$Название"},
```

```
{$project:{_id:0, "Тип":1, "Количество":{$add:
[1]}}});
```

Применение операции `$project` к каждому из документов без изменения их количества приводит к следующему результату:

```
{"result": [
{"Тип":"Художественная литература", "Количество":1},
{"Тип":"Фантастика", "Количество":1},
{"Тип":"Фантастика", "Количество":1},
{"Тип":"Фантастика", "Количество":1}], ok": 1}
```

4. Следующий шаг — группировка документов по значениям атрибута "Тип" с использованием оператора `$sum` для суммирования атрибута "Количество" для документов в группе:

```
db.Writer.aggregate ({$match:{Страна:/Рос/}},
{$unwind: "$Название"}, {$project: {_id:0,
"Тип":1, "Количество": {$add: [1]}}}, {$group:
{_id: "$Тип", "Число книг": {$sum: "$Количество"}}});
```

В результате все документы объединяются в две группы:

```
{"result": [
{"_id":"Фантастика", "Число книг":3},
{"_id":"Художественная литература", "Число
книг":1}], "ok": 1}
```

5. Последний шаг — сортировка по возрастанию числа книг в типе:

```
db.Writer.aggregate ({$match:{Страна:/Рос/}},
{$unwind: "$Название"},
{$project:{_id:0, "Тип":1, "Количество":{$add:
[1]}}},
{$group "$Тип", "Число книг": {$sum: "$Количество"}}},
{$sort: {"Число книг": 1}});
```

Результат всей цепочки агрегации:

```
{"result": [
{"_id":"Художественная литература", "Число книг":1},
{"_id":"Фантастика", "Число книг":3}], "ok":1}
```

5.12. Хранимые функции базы MongoDB

В MongoDB для хранения общих программ обработки данных предусмотрена специальная системная коллекция `system.js`. Функция записывается на языке JavaScript [15] и оформляется в виде JSON-документа в коллекции `system.js`. Добавление функций в коллекцию выполняет метод `save: db.system.js.save ()`, в котором определение функции задается параметрами:

```
db.system.js.save ({_id: "<Идентификатор функции>",
value: function <Имя функции>, (<Параметры функции>)
{<Код функции> return <Выражение — значение функции>;}});
```

Сохраненную в БД функцию можно использовать в любом коде JavaScript. Тип функции определяется типом результата, формируемого оператором `return` в теле функции.

Для выполнения на сервере JS-функции из оболочки Mongo используется метод БД `db.eval (<Выполняемая функция>, <Параметр1>, ...)`. Первым параметром задается имя функции. Последующие параметры задают фактические аргументы, используемые при выполнении функции. Для функции, хранящейся в базе, первым параметром необходимо записать запрос, выбирающий функцию из БД.

Рассмотрим пример использования методов `save ()` и `eval ()`. Сначала сохраняем в базе JS-функцию, вычисляющую, например, процентное отношение двух чисел:

```
db.system.js.save ({_id: "My%relation",
value: function MyFunc (x, y){return x/y
*100;}});
```

Атрибут `_id` используется для уникальной идентификации функции при ее выполнении, атрибут `value` содержит определение функции. Функция названа `MyFunc`, но могут быть использованы и анонимные функции. Функция `MyFunc` имеет два формальных аргумента `x` и `y` и возвращает число.

Вызов функции методом `eval`: `db.eval (db.system.js.findOne ({_id:"My%relation"}).value, 2.52, 5.0);` вернет результат 50.4.

При использовании функций важно иметь в виду, что выполнение хранимой функции блокирует доступ к серверу.

Рассмотрим пример функции, выполняющей обработку документов.

Пусть в коллекции для писателей названия их произведений хранятся в массиве "Книги". Например,

```
{"Автор":"Илья Ильф", "Книги":
[{"Название":"Золотой теленок", "Год": 1931},
{"Название":"12 стульев", "Год": 1928}]}
{"Автор": "Евгений Петров", "Книги":
[{"Название":"Золотой теленок", "Год": 1931},
{"Название":"12 стульев", "Год": 1928}]}
```

Необходимо создать в БД функцию, которая возвращает количество авторов для книги, название которой задается параметром. Создадим анонимную функцию в документе с идентификатором "`AuthorsCount`". Параметр для передачи названия книги — `Name`. В функции используется конвейерная обработка коллекции `authors`.

```
db.system.js.save ({_id: "AuthorsCount",
value: function (Name) {
a =db.authors.aggregate ({$match:{"Книги.Название": Name}},
```

```
  {$project: {"_id":1, "Количество": {$add:
[1]}}},
  {$group:{_id:"_id","Число_
авторов":{$sum:"$Количество"}})
  return a.result [0].Число_авторов}}).
```

Вызов функции `AuthorsCount`:

```
db.eval (db.system.js.findOne ({_
id:"AuthorsCount"}).value, "12 стульев");
```

вернет число 2.

5.13. Создание и использование ссылок в базе MongoDB

Возможность создавать ненормализованные документы сложной структуры, содержащие вложенные документы и/или их массивы, позволяет адекватно представлять в базе отдельные объекты, имеющие сложную структуру. Для представления взаимодействий между объектами используются ссылки. Ссылка — атрибут в документе, значение которого используется для доступа к другому документу. Ссылки являются средством прямого представления связей и используются для навигационной обработки документов [16].

MongoDB поддерживает два типа ссылок. Первый тип — «ручные» ссылки. Для однозначной идентификации документа в MongoDB ручная ссылка содержит три атрибута: `$db:`*<Имя базы>*, `$ref:`*<Имя коллекции>*, `$id:`*<Ключ документа>*. В атрибуте `$db` указывается имя базы, которая содержит ссылочный документ. При обращении к документу ссылка `$db` необязательна. При ее отсутствии поиск документа ведется в установленной базе. Атрибуты `$ref` и `$id` определяют коллекцию и идентификатор ссылочного документа в коллекции. Для обращения к документу по хранящейся в БД ссылке необходимо сначала найти исходный документ и извлечь атри-

буты ссылки, а затем использовать их в новом запросе к ссылочному документу.

Пусть создана БД, которая содержит информацию о кинофильмах в коллекции `Films` (Фильмы), данные о писателях (сценаристах) в коллекции `Writers` (Писатели), а сведения об их произведениях размещены в документах коллекции `Books` (Книги). Связи данных показаны на рис. 5.6. В документе о фильме предусмотрен атрибут-ссылка "Сценарий", указывающий на книгу, по которой поставлен фильм. В документе, представляющем автора, предусмотрен массив ссылок на книги, написанные этим автором.

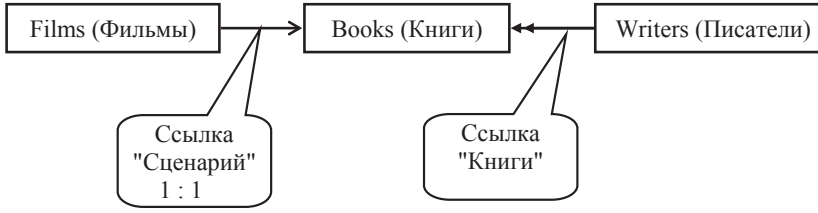


Рис. 5.6. Связи данных

Рассмотрим способ создания и использования ручных ссылок на примере ссылки "Сценарий". Пусть коллекция `Films` содержит информацию о фильме "Ночной дозор":

```
{"_id": ObjectId ("55c8214cb7f9469326e345cb"),
"Название": "Ночной дозор", "Режиссер": "Бекмамбетов Т.Н.", "Год": 2004}
```

В коллекцию `Books` помещаем сведения о романе "Ночной дозор", на основе которого был снят фильм.

```
{"_id": ObjectId ("55c82635b7f9469326e345cc"),
"Название": "Ночной дозор", "Год": 1998, "Тип":
"Фантастика"}
```

В соответствии со схемой данных (рис. 5.6) для представления сведений о книге в записи о фильме следует создать ссыл-

ку на роман "Ночной дозор". Добавим в фильм "Ночной дозор" атрибут "Сценарий", содержащий ссылку на роман "Ночной дозор" в коллекции Books.

```
db.Films.update ({"Название":"Ночной дозор"},
{$set:{"Сценарий":{"$ref": "Books",
"$id": ObjectId ("55c82635b7f9469326e345
cc")}}});
```

Проверка добавленной ссылки: запрос db.Films.find () вернет измененную запись о фильме:

```
{"Год":2004, "Название":"Ночной дозор",
"Режиссер": "Бекмамбетов Т.Н.",
"Сценарий": DBRef ("Books", ObjectId
("55c82635b7f9469326e345cc")),
"_id": ObjectId ("55c8214cb7f9469326e345cb")}
```

Обратите внимание: значение ссылки в атрибуте "Сценарий" теперь представлено функцией DBRef (…).

Для проверки ссылки найдем роман, по которому поставлен фильм "Ночной дозор". Поиск выполняется в два этапа.

1. Сначала надо получить ссылку. Для этого извлекаем из документа "Ночной дозор" атрибут "Сценарий", содержащий ссылку на книгу и сохраняем в переменной "Ссылка":

```
var Ссылка = db.Films.findOne ({"Название":
"Ночной дозор"}, {"Сценарий": 1, _id: 0});
```

Получаем в переменной "Ссылка": {"Сценарий":

```
DBRef ("Books", ObjectId
("55c82635b7f9469326e345cc"))}
```

2. Затем выбираем книгу по хранящейся в переменной ссылке:

```
db [Ссылка.Сценарий.$ref].findOne
({"_id": (Ссылка.Сценарий.$id)});
```

Перед выполнением команда findOne модифицируется подстановкой из атрибута-ссылки $ref имени коллекции, а из атрибута $id — ключа документа таким образом, что выполняется метод: db.Books.findOne ({"_id": ObjectId

("55c82635b7f9469326e345cc")}), который возвращает сведения о книге:

```
{"_id": ObjectId ("55c82635b7f9469326e345cc"),
"Название":"Ночной дозор","Год":1998,
"Тип":"Фантастика"}
```

Второй способ создания ссылок — использование для атрибута специального типа DBRef, содержащего полную ссылку на документ в текущей или другой коллекции. Атрибут типа DBRef содержит название коллекции и значение ключа (_id) документа, на который указывает ссылка. Опционально в ссылку можно включить и имя базы данных, в которой находится коллекция, содержащая ссылочный документ. В MongoDB значение ссылки задается функцией DBRef ('*<Коллекция>*', *<id документа>*, [*<Имя БД>*]).

Для получения документа по ссылке типа DBRef используется метод fetch (). Обращение *<Атрибут-ссылка типа DBRef>*.fetch () разрешает ссылку — дает прямой доступ к документу, на который указывает ссылка.

Так как при создании ручной ссылки

```
{"$ref":"Books","$id": ObjectId
("55c82635b7f9469326e345cc")}}
```

в атрибуте "Сценарий" ему был задан тип DBRef, доступ к документу по этой ссылке также возможен методом fetch (): Ссылка.Сценарий.fetch ().

Применение DBRef-ссылок продемонстрируем на примере создания связей между документом о писателе Лукьяненко и его книгами. Для этого в коллекции Writer сначала создадим документ с общими данными о писателе Лукьяненко С. В.:

```
{"_id": ObjectId ("55c8380fb7f9469326e345cf"),
"Автор": "Лукьяненко С.В.", "Страна":"Россия",
"Год_рожд":1968, "Награды": ["Аэлита", "Сталкер"]}.
```

В соответствии со схемой данных (рис. 5.6) для создания ссылок от автора на его книги добавляем в документ для автора

Лукьяненко С. В. атрибут-массив "Книги", содержащий ссылку на роман "Ночной дозор" в коллекции Books.

```
db.Writer.update ({Автор:"Лукьяненко С.В."},
{$set:{Книги: [
DBRef ("Books", ObjectId ("55c82635b7f9469326e345cc"))]}});
```

Получим запись о Лукьяненко с массивом "Книги", имеющим одну ссылку:

```
{"Автор": "Лукьяненко С.В.", "Год_рожд": 1968, "Книги": [DBRef ("Books", ObjectId ("55c82635b7f9469326e345cc"))], "Награды": ["Аэлита", "Сталкер"], "Страна":"Россия",
"_id": ObjectId ("55c8380fb7f9469326e345cf")}
```

Для демонстрации работы с массивами ссылок в коллекцию Books к книге с названием "Ночной дозор" добавим еще две книги Лукьяненко:

```
{"_id": ObjectId ("55c83596b7f9469326e345cd"), "Название": "Дневной дозор", "Год": 1999, "Тип": "Фантастика"}
{"_id": ObjectId ("55c83596b7f9469326e345ce"), "Название": "Застава", "Год": 2013, "Тип":"Внеземная фантастика"}.
```

Таким образом, в коллекции Books будет три книги автора Лукьяненко.

Для создания ссылок на три книги этого автора добавляем (через модификатор $pushAll) в массив "Книги" для автора Лукьяненко С. В. еще две ссылки:

```
db.Writer.update ({Автор:"Лукьяненко С.В."},
{$pushAll:
{Книги: [DBRef ("Books", ObjectId ("55c83596b7f9469326e345cd")),
DBRef ("Books", ObjectId ("55c83596b7f9469326e345ce"))]}});
```

Получим документ для автора Лукьяненко С. В., содержащий 3 ссылки:

```
{"Автор": "Лукьяненко С.В.","Год_рожд":1968,
"Книги":
[DBRef ("Books", ObjectId
("55c82635b7f9469326e345cc")),
DBRef ("Books", ObjectId
("55c83596b7f9469326e345cd")),
DBRef ("Books", ObjectId
("55c83596b7f9469326e345ce"))],
"Награды": ["Аэлита", "Сталкер"], "Страна":
"Россия",
"_id": ObjectId ("55c8380fb7f9469326e345cf")}
```

Теперь наличие ссылок позволяет переходить от документа автора в коллекции `Writer` к документу о книге автора в коллекции `Books`. Например, выбор первой книги Лукьяненко С. В. выполняет запрос:

```
db.Writer.findOne ({Автор:"Лукьяненко С.В."}).
Книги [0]
.fetch ();
```

Запрос вернет

```
{"_id": ObjectId ("55c82635b7f9469326e345cc"),
"Название":"Ночной дозор", "Год": 1998,
"Тип":"Фантастика"}
```

Для получения всех книг Лукьяненко необходим перебор по ссылкам в массиве `Книги`. Для этого сначала определяем число ссылок в массиве `Книги` и сохраняем в переменной `Numer`:

```
var Numer = db.Writer.findOne (
{Автор: "Лукьяненко С.В."}).Книги.length;
```

Объявляем массив `Docs` для хранения найденных документов о книгах:

```
var Docs = [];
```

Выбираем данные о книгах Лукьяненко и сохраняем в массиве Docs

```
for (var i=0; i < Numer; i++)
{Docs [i]=db.Writer.findOne ({Автор:"Лукьяненко
С.В."}).
Книги [i].fetch ()};
```

Для вывода в консоль Mongo Shell отдельных атрибутов из документов, найденных по ссылкам, используется функция print:

```
for (var i = 0; i < Numer; i++) {
print (db.Writer.findOne ({Автор:"Лукьяненко
С.В."})
.Книги [i].fetch ().Название);}
```

Из коллекции Books будут выведены названия книг Лукьяненко С. В.

6. Большие данные

Большие данные в информационных технологиях — серия подходов, инструментов и методов обработки структурированных и неструктурированных данных огромных объемов и значительного многообразия для получения воспринимаемых человеком результатов, эффективных в условиях непрерывного прироста, распределения по многочисленным узлам вычислительной сети. К этому, пожалуй, следует добавить, что в отличие от бизнес-аналитики корпоративных данных предметом обработки больших данных является цифровая информация, сформированная множествами действующих в Интернете стационарных и мобильных устройств. Поэтому содержание информации описывает состояние значительно большей частей общества, а значит, и результаты обработки затрагивают интересы больших масс людей.

В информационных технологиях признаками больших данных служат:

- объем данных, который оказывается настолько велик, что не позволяет выполнить хранение и обработку традиционными методами и средствами;
- быстрый рост объема генерируемой информации;
- разнообразие используемых форматов неструктурированных и моделей для структурированных данных;
- одинаковая ценность всех данных для решения задачи, т. е. из всей массы априорно не выделяются несущественные данные, которые можно бы не хранить и не обрабатывать.

Объем и высокая скорость генерации данных обусловлены массовостью источников информации: люди и машины, работающие в Интернете, также все открытые информационные системы. Разнообразие форм представления данных обусловлено разными типами источников, формирующих данные как в структурированном, так неструктурированном виде с использованием разных форматов. При этом основная масса данных является неструктурированной.

В сравнении с классическими базами при обработке больших данных допускается избыточность и неконсистентность данных. Например, временная несогласованность данных или несоответствие реплики основным данным источника. Подобная «неточность» данных допустима в силу того, что целью обработки является не вычисление новых данных, а обнаружение статистических зависимостей и трендов развития процессов.

В технологии обработки традиционных баз данных создано отдельное направление — бизнес-аналитика (BI, Data Mining) — и разработано немало «коробочных» продуктов для пакетной обработки данных, например STATISTICA Data Miner (компании StatSoft), SPSS (SPSS, Clementine), Institute (SAS Enterprise Miner), Megaputer Polyanalyst Suite (Megaputer software), IBM Intelligent Miner for Data и др. Технология применения этих продуктов требует сосредоточенной в одном центре базы и процесса обработки данных и поэтому не позволяет использовать их для хранения и обработки больших данных.

Потребность достижения приемлемой скорости обработки больших данных стимулировала появление новых методов, ориентированных на использование массово-параллельных вычислений. Влияние распараллеливания вычислений на время обработки определяется законом Амдала: «Суммарное время выполнения задачи на параллельной системе не может быть меньше времени выполнения самого длинного последовательного фрагмента».

Ускорение времени решения задачи на n процессорах, по сравнению с однопроцессорным решением: определяется формулой:

$$S_n = \frac{1}{\lambda + \frac{1-\lambda}{n}},$$

где S_n показывает, во сколько раз уменьшится время обработки на n процессорах в сравнении с однопроцессорным вычислением; λ — доля последовательных расчётов в общем объеме вычислений; $(1 - \lambda)$ — доля вычислений, которая может быть распараллелена идеально (время вычисления будет обратно пропорционально числу задействованных узлов). При полном распараллеливании $\lambda \to 0$, $S_n \to n$.

Например, если $\lambda = 0{,}25$ — четверть вычислений выполняется последовательно (не распараллеливаются). Тогда сокращение времени обработки в зависимости от числа параллельных процессов, представленное на рис. 6.1 [17], показывает, что при $n = 10$ фактически наступает насыщение и дальнейшее увеличение числа процессоров не сокращает время обработки.

$\lambda = 0{,}25$
$n = 2$, $S_n = 1{,}6$;
$n = 5$, $S_n = 2{,}5$;
$n = 10$, $S_n = 3,08$; — Насыщение
$n = 20$, $S_n = 3,5$;
$n = 40$, $S_n = 3,7$;
$n = 50$, $S_n = 3,77$;

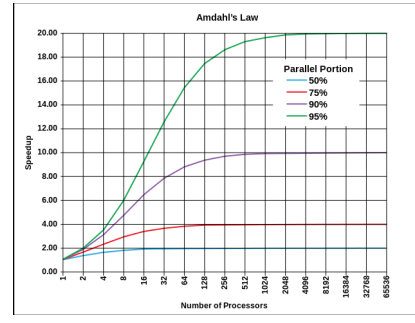


Рис. 6.1. Зависимости времени обработки от числа процессов

Однако особенность больших данных, заключающаяся не только в их массовости, но и однородности, позволяет выполнять распараллеливание процесса обработки на сотни и ты-

сячи узлов. Для реализации таких процессов необходимы средства отказоустойчивого распределенного хранения огромных объемов данных и организации надежных параллельных вычислений с последующим слиянием их результатов.

Представленной задаче наилучшим образом соответствуют данные, накопленные в Интернете. Они исходно размещены на разных узлах в дата-центрах и могут обрабатываться параллельно. При этом используемая технология обработки должна учитывать следующие требования.

1. Данные хранятся на многих серверах, и, чтоб избежать больших объемов передач данных, их надо обрабатывать в местах хранения. Вместо традиционной передачи данных в узлы обработки необходима передача программ в места хранения данных. Серверы хранения должны быть и серверами обработки данных.

2. Неизбежные отказы серверов хранения и обработки требуют реализации их автоматического восстановления. В целях достижения общей эффективности необходима балансировка и перераспределение нагрузки между узлами обработки.

3. Реализация параллельных вычислений требует программирования и синхронизации параллельных процессов для большого и меняющегося числа параллельных узлов.

Возможность автоматически накапливать огромные массивы информации из различных источников и наличие финансовых стимулов использования этих данных обусловили лидирующую роль интернет-компаний в разработке технологий анализа больших данных. Именно в этих компаниях были разработаны распределенные файловые системы в сочетании с технологией MapReduce для массово-параллельной обработки данных и первые NoSQL-хранилища.

7. Распределенные файловые системы

Распределенная файловая система (РФС) [18] позволяет множеству одновременно работающих пользователей разделять доступ к файлам, размещенным на разных машинах, объединенных в сеть. В основе распределенных файловых систем лежит модель клиент-сервер. В данном случае под клиентом понимается машина, которая обращается к некоторому файлу, а под сервером — машина, хранящая файлы и обеспечивающая к ним доступ. Некоторые системы требуют, чтобы клиенты и серверы были разными машинами, в то время как другие допускают, чтобы одна машина работала и как клиент, и как сервер.

Распределенные файловые системы должны выполнять ряд важных требований:

- сетевая прозрачность: способы доступа и обработки должны быть одинаковыми для локальных и удаленных файлов;
- прозрачность размещения: имя файла не должно определять его местоположения в сети и не должно меняться при изменении его физического размещения;
- высокая доступность: выполнение запросов клиентов за время, приемлемое для решаемых задач;
- мобильность пользователя: пользователи должны иметь возможность обращаться к разделяемым файлам из любого узла сети;
- отказоустойчивость: система должна продолжать функционировать при неисправности отдельного компонента (сервера или сегмента сети);

- масштабируемость: возможность наращивания объема хранимых данных и производительности путем добавления необходимых компонентов.

К перечисленным основным требованиям отдельные области применения могут предъявлять дополнительные требования, например шифрования данных, использования определенных операционных систем, тип лицензии и др.

В настоящее время разработано множество РФС, имеющих различную архитектуру и в разной степени соответствующих предъявляемым требованиям. В области хранения и обработки больших данных распространение получила Hadoop Distributed File System (HDFS). HDFS является клоном файловой системы GFS, разработанной компанией Google. HDFS создана с согласия Google в проекте Apache Software Foundation, предоставляющем свободную лицензию на свои продукты. В значительной мере популярность HDFS обусловлена ее использованием в технологии обработки больших данных MapReduce, также реализованной в проекте Apache Software. Принципы построения HDFS и технологии MapReduce рассматриваются в последующих разделах.

7.1. Hadoop Distributed File System (HDFS)

HDFS подобно традиционным файловым системам поддерживает древовидную структуру каталогов и основанную на правах модель доступа к файлам.

Система хранения и доступа оптимизирована под однократную запись и многократное последовательное чтение больших (в десятки гигабайт) файлов. Именно такая обработка данных преобладает в технологии MapReduce. Размещение файлов выполняется блоками (>64MB) в большом кластере недорогих ненадежных устройств. Для обеспечения отказоустойчиво-

сти блоки файла реплицируются на разные устройства. Размер блока и репликационный фактор настраиваются для каждого файла. Особенности HDFS в значительной степени обусловлены ее ориентацией на хранение и обработку больших данных с использованием технологии Map/Reduce. HDFS разработана на языке Java, может быть развернута на широкий спектр машин.

Архитектура HDFS и типичный Hadoop-кластер

Hadoop-кластер имеет иерархическую двухуровневую структуру и состоит из узлов трех типов: NameNode, Secondary NameNode, DataNode [19].

Обычно HDFS-кластер содержит один сервер метаданных NameNode, который хранит все метаданные и управляет пространством имен файловой системы. Пользовательские приложения получают доступ к файловой системе с помощью клиента HDFS. Сервер метаданных принимает запросы клиента на чтение и запись файлов, а также выполняет все операции с областью имен файловой системы: открытие, закрытие и переименование файлов и каталогов. Файл разбивается на один или несколько блоков, эти блоки хранятся в наборе DataNode-серверов. Обычно в каждом физическом узле кластера размещается один DataNode-сервер. DataNode выполняют создание, удаление и репликацию блоков по команде NameNode. Соответствие между файлами и их блоками в узлах DataNode хранит NameNode.

Каждый DataNode регулярными сообщениями информирует NameNode о своей работоспособности и систематически обновляет информацию обо всех хранимых блоках для поддержания нужного уровня их репликации. Архитектура типового кластера HDFS показана на рис. 7.1.

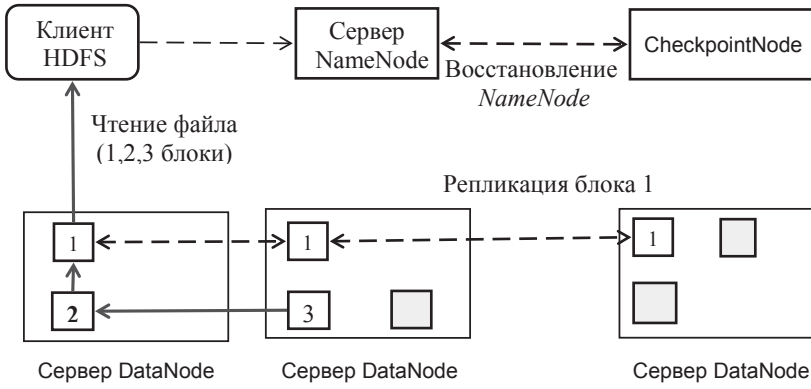


Рис. 7.1. Архитектура HDFS-кластера

На рис. 7.1 читаемый файл состоит из трех блоков. Для первого блока показаны две реплики, находящиеся на отдельных серверах данных. Запись файла выполняется следующим образом.

1. Клиент разрезает файл на цепочки блоков.
2. Клиент обращается к NameNode операцией записи, сообщая количество блоков и требуемое число реплик.
3. NameNode возвращает клиенту список адресов DataNode.
4. Клиент соединяется с первым DataNode из списка. Если не получилось соединиться с первым, то соединяется со вторым и т. д. Если не получится соединиться ни с одним из серверов списка, выполняется откат записи.
5. Клиент выполняет запись первого блока на первый DataNode, первый — на второй и т. д. по числу заданных реплик.
6. По окончании записи последней реплики в обратном порядке ($4 \rightarrow 3$, $3 \rightarrow 2$, $2 \rightarrow 1$, $1 \rightarrow$ клиенту) присылаются сообщения об успешной записи.
7. Первый DataNode оповещает NameNode об успешной записи блока. NameNode формирует новый список DataNode для записи второго блока и т. д.

Сервер метаданных NameNode хранит образ пространства имен в оперативной памяти. Копия образа, хранящаяся в локальной файловой системе сервера метаданных, называется контрольной точкой. Данные текущих изменений HDFS перед произведением самих изменений сохраняются в журнальном файле в локальной файловой системе. Для обеспечения отказоустойчивости поддерживается несколько синхронно обновляемых копий этих файлов на множестве независимых локальных разделов и удаленных NFS-серверах. Сервер метаданных в HDFS может использоваться в одном из двух дополнительных режимов: сервера файлов контрольных точек (CheckpointNode) и сервера резервных копий (BackupNode). Режим работы устанавливается в процессе загрузки сервера. Сервер файлов контрольных точек периодически объединяет данные из файла контрольной точки и журнала для создания нового файла контрольной точки и очистки журнала. Этот сервер обычно устанавливается на отдельном узле. В случае отказа для восстановления сервера NameNode используется файл контрольной точки, хранящийся в CheckpointNode. Наличие поблочных реплик обеспечивает необходимую отказоустойчивость хранения и эффективность чтения больших объемов накапливаемых и неизменяемых данных.

8. Технология MapReduce

Являясь пионером в области Больших данных, компания Google разработала для внутреннего применения ряд продуктов для хранения и обработки таких данных: Big Table, Google File System (GFS) и технологию MapReduce. Причем GFS и MapReduce образуют комплекс, в котором GFS обеспечивает хранение и эффективный доступ, а MapReduce — обработку больших данных. При содействии компании Yahoo! с согласия Google, используя их идеи и многие архитектурные решения, в проекте Apache Software Foundation создали программное обеспечение Apache Hadoop с открытым кодом, которое включает Hadoop Distributed File System и Hadoop MapReduce (HDMR). Архитектура, решения по организации и доступу к данным в HDFS рассмотрены в главе 7. Рассмотрим модель и технологию обработки данных в Hadoop MapReduce.

В MapReduce реализована идея параллельной обработки списков, используемая в функциональных языках. Исходные данные представляются в форме однородного списка из элементов вида <***ключ***> или пар <***ключ***, ***значение***>. Список фрагментирован и распределен по узлам вычислительной сети. Обработка исходного списка выполняется в два этапа.

1 этап. Функция Map: ***S*** = **Map (*F, C*)**. Создает новый список ***S,*** применяя заданную пользователем функцию ***F*** к входному списку ***C***. Функция ***F*** для каждого элемента входного списка формирует пару: <***ключ***>:<***значение***>.

2 этап. Функция Reduce: V = **Reduce** (***G***, ***S***). Выполняет свертку (группировку) ***G*** элементов списка ***S.*** Пользовательская функция ***G*** объединяет пары входного списка ***S*** по определенным значениям ключа в результирующий список ***V***.

Обычное применение метода MapReduce — получение из исходного массива информации статистических данных в форме списка из элементов следующего вида: <***данное — ключ***> — <***значение, соответствующее ключу*** >.

Например, <***название товара***> -<***общее количество или стоимость товара***>.

Рассмотрим работу метода на одном из его первых применений — оценка частоты слов в запросах к поисковым машинам Интернета. Частота слов содержит информацию о том, что интересует группы людей и общество в целом.

Исходный набор — список слов (ключей) в запросе, для которых выполняется обработка.

1 шаг. Для подсчета частоты слов пользовательская функция ***Map*** порождает множество пар <***ключ эл-та, значения эл-та***> в виде <***слово, 1***>, где «***слово***» есть ключ элемента, а **1** — значение элемента. При этом каждый запрос может обрабатываться независимо и параллельно. Например, подсчет частоты слов в текстах трех запросах, находящихся в разных узлах:

скачать бесплатно | смотреть фильмы | скачать музыку бесплатно

Результат обработки функцией Map:

<скачать, 1> <бесплатно, 1> | <смотреть, 1> <фильмы, 1> | <скачать, 1> <музыку, 1> <бесплатно, 1>

2 шаг. Список полученных пар обрабатывается процедурой Reduce, которая по заданному правилу (функции) объединяет (уменьшает, сворачивает, группирует) результаты отдельных Map-функций по значениям ключа. Для списков <***слово, 1***>, ... функция объединяет элементы списка с одинаковыми ключами-словами и суммирует единицы. В результате на выходе ока-

зывается список всех исходных ключевых слов с количеством их упоминаний в запросах.

После обработки функцией на шаге Reduce получим:

<скачать, 2> <бесплатно, 2>| <смотреть, 1> <фильмы, 1> <музыку, 1>

Другой пример. Биллинговая система учета телефонных соединений.

На серверах в файлах хранятся записи о телефонных соединениях абонентов.

Структура записи:

<Id записи>, <исходящий номер>, <номер вызываемого абонента>,

<момент соединения>, <момент разъединения>.

Цель обработки: подсчитать общее время исходящих звонков для каждого номера. Исходящий номер служит ключом, для которого выполняется обработка списка записей. Пусть в списке находятся три исходных записи заданной структуры:

001, 111-123-1234, 555-777-7777, 23.08.15 08:02:14, 23.08.15 08:04:24

002, 111-123-1234, 333-888-8888, 23.08.15 10:10:22, 23.08.15 10:13:24

003, 222-333-1234, 999-777-7777, 23.08.15 18:10:22, 23.08.15 18:11:34

Функция на шаге Map для каждого соединения вычисляет время разговора:

001, 111-123-1234, 00:02:10,

002, 111-123-1234, 00:03:03,

003, 222-333-1234, 00:01:12

На шаге Reduce выполняется суммирование длительностей разговоров для каждого ключа. В результате получим:

111-123-1234, 00:05:13

222-333-1234, 00:01:12

Из примеров видно, что процессы обрабатывают наборы данных по одной схеме, отличаясь способом преобразования дан-

ных (функциями) на этапах Map и Reduce. Общая схема обработки данных представлена на рис. 8.1.

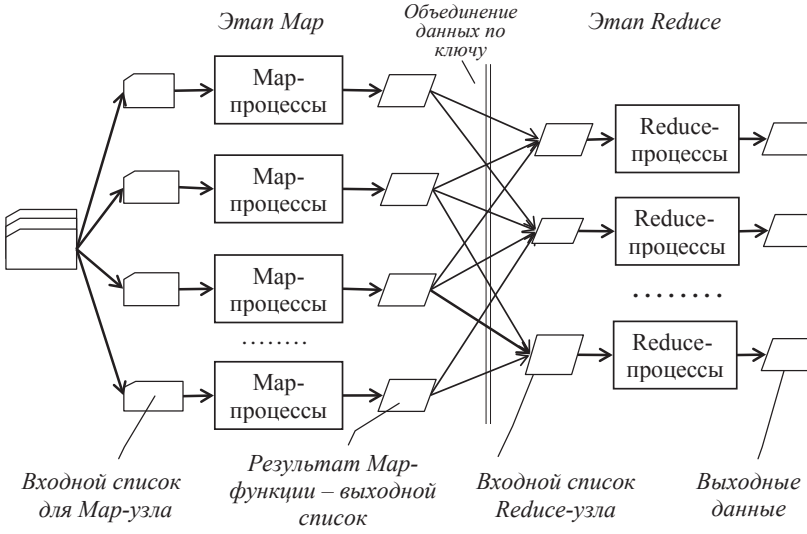


Рис. 8.1. Схема обработки данных MapReduce

Из этой схемы возникла идея разделять программы обработки списков данных на предметную и обеспечивающую части. Предметная часть содержит требуемую обработку данных и реализуется двумя функциями, создаваемыми пользователем для этапов Map и Reduce. Обеспечивающая часть — каркас-программа (технология), которая выполняет хранение и параллельную обработку пользовательскими функциями исходных массивов данных, а также реализует все служебные задачи: масштабирование, балансировку нагрузки и компенсацию отказов на тысячах узлов вычислительной сети.

8.1. Архитектура Hadoop MapReduce

Hadoop MapReduce имеет двухуровневую архитектуру взаимодействия единственного управляющего (JobTracker) и множества исполняющих (TaskTracker) процессов [20]. JobTracker и TaskTracker используют данные, хранящиеся в HDFS, встроенной в архитектуру HDMR:

- JobTracker выполняется на узле NameNode в HDFS;
- процессы TaskTracker независимо выполняются на узлах DataNode;
- процессы TaskTracker на этапе Map обрабатывают данные своего узла, а на этапе Reduce — данные своего узла и максимально близких узлов DataNode. Архитектура Hadoop MapReduce представлена на рис. 8.2.

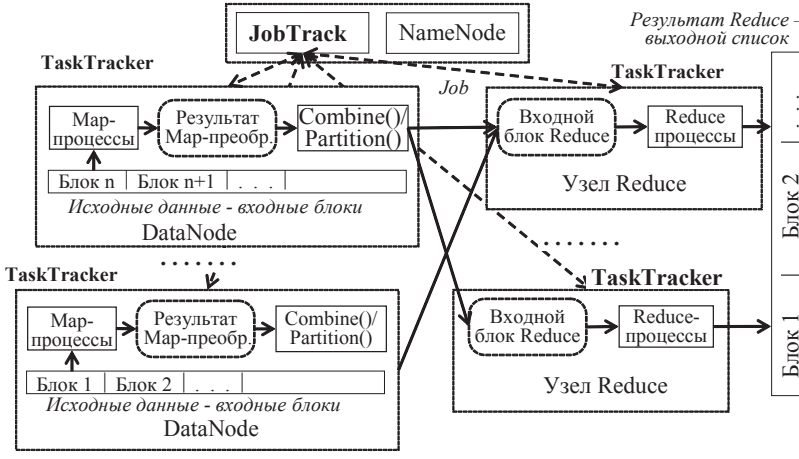


Рис. 8.2. Архитектура Hadoop MapReduce

Программный клиент взаимодействует только с JobTracker по следующей схеме: полученное от клиента задание (Job) JobTracker разбивает на множество Map-задач и множество

Reduce-задач. Используя информацию NameNode о количестве и размещении файловых блоков исходных данных, JobTracker вычисляет число задач, которые необходимо создать в узлах TaskTracker. Каждый узел TaskTracker получает от JobTracker список его задач, загружает и выполняет их коды, обрабатывая назначенные узлу блоки данных. Периодически в процессе обработки TaskTracker отсылает JobTracker состояние решаемых задач.

Для управления процессами MapReduce JobTracker выполняет следующие функции:

- планирование Map- и Reduce- заданий и промежуточных сверток в узлах TaskTracker;
- координация и мониторинг заданий;
- переназначение завершившихся неудачей заданий другим узлам TaskTracker.

В свою очередь, TaskTracker выполняет следующие функции:

- Map- и Reduce-задания;
- контроль исполнения заданий;
- отправка сообщений JobTracker о статусе задачи и завершении работы;
- отправка диагностических сообщений узлу JobTracker.

В Hadoop MapReduce используются аналогичные HDFS средства восстановления процессов. При сбое узла TaskTracker узел JobTracker переназначает его задания исправному узлу TaskTracker. При отказе JobTracker-узла выполняется его перезапуск. При перезапуске JobTracker читает данные о последней контрольной точке (checkpoint), восстанавливает свое состояние и продолжает работу с момента создания контрольной точки.

8.2. Преимущества и недостатки Hadoop MapReduce

Актуальность задач по выявлению тенденций и прогнозированию трендов развития процессов на основе автоматически накапливаемых массовых данных обусловили распространение

Hadoop MapReduce во многих сферах деятельности. Широкое применение в сочетании с открытостью кода привели к созданию собственной эко системы для Hadoop MapReduce, включающей следующие продукты:

1. Хранилище Apache Hive со средствами работы с данными, хранящимися в файлах в Apache HDFS или Apache HBase. Возможность наложения на хранимые данные структуры привела к созданию простого SQL-подобного языка запросов — Hive QL. Аналитические возможности языка расширяются подключением в запросы пользовательских функций Map и Reduce.

2. Framework Apache Pig. Apache Pig создан для решения MapReduce-задач с помощью процедурного языка Pig Latin. Программы на Pig Latin обрабатывают данные, представленные в формате записей с упорядоченным набором полей с доступом по индексу.

Популярность MapReduce обусловила включение идей этой технологии в системы управления БД. Так в СУБД MongoDB создан метод MapReduce, последовательно применяющий две пользовательские функций (Map и Reduce) для обработки коллекции документов. Результаты обработки могут быть сохранены в новой коллекции базы для последующей обработки.

При использовании Hadoop MapReduce необходимо иметь в виду, что исходная ориентация на большие данные ограничила область эффективного применения технологии задачами обработки однородных массивов с объемом от 100 Гб в вычислительной системе, состоящей из сотен узлов.

8.3. Реализация Map/Reduce в MongoDB

В MongoDB технология обработки данных MapReduce реализована методом коллекции документов [21]. Обрабатываемая коллекция может находиться на одном сервере или быть

распределенной в кластере MongoDB-серверов. Каждый рабочий MongoDB-сервер — шард (shard) кластера — содержит и независимо обрабатывает свою часть коллекции. Управление и взаимодействие рабочих серверов обеспечивает маршрутизатор кластера mongos. Форма обращения приложения к методу MapReduce одинакова как для одиночного кластера, так и для кластера MongoDB-серверов. При работе в кластере приложение, запускающее метод MapReduce, должно быть соединено с маршрутизатором (mongos), который распределяет и контролирует выполнение задания на шардах кластера.

Вызов метода Map-Reduce для обработки коллекции имеет три параметра и выглядит следующим образом:

```
db.<Обрабатываемая коллекция>.mapReduce (
<Функция map>,
<Функция reduce>,
{out: <Коллекция — результат обработки>,
[query: <Селектор>,
sort: <Атрибуты для сортировки результата>,
limit: <Число документов>,
finalize: <Функция дополнительной обработки результата свертки>]})
```

<Функция map> — JavaScript-функция, которая независимо обрабатывает каждый документ коллекции, создавая на выходе пару: *<Ключ>*, *<Значение>*. Структура функции `map`:

```
var <Имя map-функции> = function () {.... emit (<Ключ>, <Значение>);};
```

В теле функции задается алгоритм вычисления ключа и значения через атрибуты документа. В функции доступ к атрибуту обрабатываемого документа выполняется через обращение `this.`*<Имя атрибута>*. Генерацию пары *<Ключ>*, *<значение>*. выполняет функция `emit`, имеющая два параметра: первым параметром указывается ключ, вторым — соответствующее ключу значение. Значение может быть и документом.

Функции reduce с каждым значением ключа передается массив данных, созданных map-функцией значений для этого ключа. Прототип функции reduce:

```
var <Имя функции reduce> = function (<Ключ>,
<Значения для ключа>) {......
return <Выражение для «свертки» значений>;};
```

Функция реализует «свертку» множества значений, соответствующих конкретному ключу в одно (обычно агрегатное) значение. Результат свертки определяет выражение в операторе return, заканчивающее обработку ключа и возвращающего результат в выходную коллекцию в виде документа *{<Ключ>,< Результат «свертки» значений для ключа >}*.

Параметр out: *<Коллекция — результат обработки>* содержит имя коллекции, в которой сохраняется результат полной обработки. Если такая коллекция существует, то по умолчанию она будет заменена. При значении параметра out: {inline: 1} результат передается в консоль mongo.exe.

Необязательные параметры:

1) query: *<Селектор>* содержит селектор для запроса, выбирающего документы из входной коллекции для обработки map-функцией;

2) sort: *<Атрибуты для сортировки>* задает сортировку входных документов; предварительная сортировка документов по ключу генерируемой в map-функции пары *<Ключ>*, *<Значение>* уменьшает число выполняемых операций; входная коллекция должна быть проиндексирована по этому ключу;

3) limit: *<Число документов>* ограничивает количество входных документов, обрабатываемых map-функцией;

4) finalize: *< Функция дополнительной обработки>* — необязательная JavaScript-функция обработки коллекции, сформированной функцией reduce. Прототип функции finalize:

```
function (Ключ, «Свернутое значение») {.......
return <Обработанный документ>}
```

Функции reduce и finalize обрабатывают только подаваемые на вход данные без обращения к базе.

Рассмотрим примеры использования MapReduce в MongoDB.

Пусть в коллекции Writer находится пять документов:

```
{"Автор": "Пелевин В.О.",
"Название": ["Чапаев и Пустота"], "Тип": "Роман"}
{"Автор": "Пелевин В.О.",
"Название": ["Бетман Аполло"], "Тип":"Фантастика"}
{"Автор": "Лукьяненко С.В.",
"Название": ["Ночной дозор", "Дневной дозор"],
        "Тип":"Фантастика"}
{"Автор": "Кант И.",
"Название": ["Критика чистого разума", "Критика практического разума"], "Тип":"Философия"}
{"Автор": "Пелевин В.О.", "Название": "Generation «П»",
"Тип": "Роман"}
```

Используем MapReduce для вычисления количества произведений каждого типа. Для этого выбираем ключом атрибут "Тип" и создаем функцию map, которая для каждого входного документа генерирует пару (key, value), где ключ (key) получает значение атрибута «Тип», а value — количество произведений — элементов в массиве "Название". При создании map-функции необходимо учесть, что в последнем документе значение атрибута "Название": "Generation «П» " не является массивом и попытка определить его длину приводит к ошибке. Поэтому в функции предусмотрена проверка типа атрибута: typeof (this.Название) == 'object'.

Функция map имеет вид:

```
var mapFunc1 = function () {
var key = this.Тип;
```

```
if (typeof (this.Название) == 'object')
        value= this.Название.length;
else value = 1;
emit (key, value);};
```

Задачей функции `reduce` является суммирование элементов (количество произведений) в массиве значений для каждого значения ключа (тип произведения):

```
var reduceFunc1 = function (keyType, valuesNumb) {
return Array.sum (valuesNumb);};
```

Последовательное выполнение обеих функций реализует метод обрабатываемой коллекции `db.Writer.mapReduce` с сохранением результата в коллекции `"TypeNumb"`:

```
db.Writer.mapReduce (
mapFunc1, reduceFunc1, {out: "TypeNumb"});
```

По окончании обработки коллекции `Writer` метод возвратит статистику решения задачи в формате документа:

```
{"result":"TypeNumb",— имя выходной коллекции — результат
"timeMillis": 37,— время решения задачи
"counts":{"input":5,— количество обработанных документов
"emit": 5,— количество пар <Ключ, Значение>,
созданных функцией mapFunc1
"reduce": 2,— количество пар <Ключ, Значение>,
сгруппированных по совпадающим ключам
"output": 3 — количество документов в TypeNumb},
"ok": 1,}.
```

Запрос `db.TypeNumb.find ({})` вернет коллекцию с результатом обработки:

```
{"_id": "Роман", "value": 2}
{"_id": "Фантастика", "value": 3}
{"_id": "Философия", "value": 2}
```

В следующем примере рассмотрим обратную задачу подсчета числа авторов для каждой книги в коллекции `Writer`. Клю-

чом служит название книги, являющееся элементом массива "Название", а значение — число документов (авторов), содержащих в атрибуте-массиве это название.

Решение подобной задачи с использованием функций и метода `aggregate` рассмотрено в разделе 5.11. Для тестирования разрабатываемых `map`- и `reduce`-функций дополним коллекцию писателей `Writer` двумя книгами, имеющими несколько авторов:

```
({"Автор": "Илья Ильф",
"Книги": [{"Название":"Золотой теленок",
"Год":1931},
{"Название":"12 стульев", "Год": 1928}]})
({"Автор": "Евгений Петров",
"Книги": [{"Название":"Золотой теленок",
"Год":1931},
{"Название":"12 стульев", "Год": 1928}]})
```

Структура новых документов отличается от введенных ранее. Теперь "`Название`" является атрибутом документа, вложенного в массив "`Книги`". Наличие атрибута "`Книги`" позволит выполнить тестирование новых `map`- и `reduce`-функций только для этих двух документов.

Создаем функцию для этапа `map`, которая проверяет в документе наличие атрибута "`Книги`" и для такого документа генерирует набор пар вида <*Ключ*, *Значение*>, где ключом является название книги, а значением 1.

```
var mapFunc2 = function ()
{if (this.Книги!= null) {//проверка наличия
атрибута "Книги"
for (var i = 0; i < this.Книги.length; i++) {
emit (this.Книги [i].Название, 1)}
} };
```

Функция `Reduce` суммирует значения 1 для каждого ключа (название книги) и записывает в виде документа {<*Ключ*>, <*Сумма единиц*>} в выходную коллекцию:

```
var reduceFunc2 = function (keyНазв, valuesКол) {
return Array.sum (valuesКол);};
```

Для выполнения функций `map` и `Reduce` с сохранением результата в коллекции `"Книга_ЧислоАвторов"` вызываем метод MapReduce:

```
db.Writer.mapReduce (mapFunc2, reduceFunc2,
{out: "Книга_ЧислоАвторов"});
```

Статистика выполнения метода:

```
{"result": "Книга_ЧислоАвторов", "timeMillis":
1465,
"counts":{"input":7, "emit":4,
"reduce":2, "output":2},
"ok": 1,}
```

Из статистики следует, что на входе `map`-функции всего было 7 документов. Проверка `this.Книги!= null` оставила два последних документа. По ним было эмитировано `("emit":4)` четыре пары вида (<*Название книги*>, 1), которые функцией `reduce` были объединены в две пары и записаны в коллекцию `"Книга_ЧислоАвторов"`.

Запрос `db.Книга_ЧислоАвторов.find ({}, {})` к полученной коллекции возвращает результат обработки:

```
{"_id": "12 стульев", "value": 2}
{"_id": "Золотой теленок", "value": 2}
```

Исключить из обработки документы, не имеющие атрибута `"Книги"`, можно было бы и с помощью селектора, заданного непосредственно в параметре метода `mapReduce`.

Для этого сначала закомментируем проверку наличия атрибута `"Книги"` в функции map:

```
var mapFunc2 = function () {
//if (this.Книги!= null)
for (var i = 0; i < this.Книги.length; i++) {
emit (this.Книги [i].Название, 1)}
};
```

Добавим в вызов метода `mapReduce` параметр `query` с селектором, выбирающим документы с атрибутом `"Книги"`:

```
db.Writer.mapReduce (mapFunc2, reduceFunc2,
{out: "Книга_ЧислоАвторов",
query: {"Книги": {$exists: true}}})
```

Метод вернет прежний результат, а в статистике выполнения метода изменится количество обработанных документов: `"input": 2`.

Возможность выполнения метода `mapReduce` для коллекции, распределенной в кластере серверов, позволяет построить эффективную обработку значительно больших объемов данных по сравнению с обычными функциями на JavaScript и конвейерной обработкой.

Список библиографических ссылок

1. Прогноз тенденций в сфере IT на 2012–2016 гг. компании Gartner // Securitylab.ru: информационный портал по безопасности. Режим доступа: http://www.securitylab.ru/news/410717.php (дата обращения: 10.09.2015).
2. Россия создает 2,4 % мирового объема данных // Cnews.ru: издание о высоких технологиях. Режим доступа: http://www.cnews.ru/news/top/rossiya_sozdaet_24_mirovogo_obema_dannyh (дата обращения: 10.09.2015).
3. Рагимова С. Большие данные (Big Data) — одна из ключевых технологий будущего // Коммерсант.ru: сайт. Режим доступа: http://www.kommersant.ru/doc/2614791?9f476940 (дата обращения: 10.09.2015).
4. Рост объема информации — реалии цифровой вселенной // Технологии и средства связи. 2013. № 1. Режим доступа: http://www.tssonline.ru/articles2/fix-corp/rost-obema-informatsii — realii-tsifrovoy-vselennoy (дата обращения: 10.09.2015).
5. Майер-Шенбергер В., Кукьер К. Большие данные : Революция, которая изменит то, как мы живем, работаем и мыслим. М. : Манн, Иванов и Фербер, 2013. 240 с.
6. Кузнецов С. Д. Объектные модели ODMG и SQL десять лет спустя: нет противоречий // Труды ИСП РАН. 2015. Т. 27, вып. 1. С. 173–192.
7. NoSQL базы данных: понимаем суть // Habrahabr.ru: сайт. Режим доступа: http://habrahabr.ru/post/152477/ (дата обращения: 10.09.2015).
8. Кузнецов С. Д. Объектно-ориентированные базы данных в стандарте ODMG // Базы данных и информационные технологии XXI века : материалы международной научной конференции. 2005. Режим доступа: http://citforum.ru/database/articles/manifests/art_28_2_2.shtml

9. Кайт Т. Oracle для профессионалов : Архитектура, методики программирования и особенности версий 9i, 10g и 11g. / пер. с англ. Н. А. Мухина ; под ред. Ю. Н. Артеменко. 2-е изд. М. : Вильямс, 2012. 848 с.
10. Фейерштейн С., Прибыл Б. Oracle PL/SQL для профессионалов. СПб. : Питер, 2005. 941 с.
11. Петрелевич С. Использование объектных типов в PL/SQL // OraHome.ru: каталог статей о СУБД. Режим доступа: http://www.orahome.ru/ora-artic/43 (дата обращения: 09.02.2015).
12. Бэнкер К. MongoDB в действии. М.: ДМК Пресс, 2012. Режим доступа: http://e.lanbook.com/books/element.php?pl1_cid=25&pl1_id=4156.
13. Буторин Д. Н. Разработка баз данных в MongoDB : учеб. пособие. Красноярск : КГПУ им. В. П. Астафьева, 2013. 236 с.
14. Концепция агрегации данных в MongoDB // MongoDB Documentation. Режим доступа: http://docs.mongodb.org/manual/core/aggregation/Aggregation Concepts (дата обращения: 09.02.2015).
15. Программирование на javascript // Javascript.ru: сайт. Режим доступа: http://javascript.ru/function (дата обращения: 09.02.2015).
16. Ссылки в базе данных MongoDB // Tutorialspoint: сайт. Режим доступа: http://www.tutorialspoint.com/mongodb/mongodb_database_references (дата обращения: 09.02.2015).
17. Закон Амдала // Википедия. Режим доступа: https://ru.wikipedia.org/wiki / Закон_Амдала (дата обращения: 09.02.2015).
18. Распределенные файловые системы // Free Info : сайт. Режим доступа: http://freekniga7.narod.ru/korpsyst/glava_13.htm (дата обращения: 09.02.2015).
19. Распределенная файловая система Hadoop/Chansler R. [et al.] // Linux по-русски: виртуальная энциклопедия. Режим доступа: http://rus-linux.net/MyLDP/BOOKS/Architecture-Open-Source-Applications/Vol-1/hdfs-2.html (дата обращения: 09.02.2015).
20. Hadoop MapReduce : Основные концепции и архитектура // Codeinstinct.pro: [блог]. Режим доступа: http://www.codeinstinct.pro/2012/08/mapreduce-design.html (дата обращения: 09.02.2015).
21. Реализация MapReduce в MongoDB // MongoDB Documentation. Режим доступа: http://docs.mongodb.org/manual/core/mapreduce/ (дата обращения: 09.02.2015).


УНИВЕРСИТЕТСКОГО КНИГОИЗДАНИЯ
УНИВЕРСИТЕТСКАЯ КНИГА
2015


9 785799 618278